TDMQ for CKafka

General References

Product Documentation

Contents

General References
Conducting Production and Consumption Pressure Testing on CKafka

Last updated：2024-01-09 15:02:47

Testing Tool

The open-source script of the Kafka client can be used for Kafka producer and consumer performance testing. Test results are displayed mainly based on the size of messages sent per second (MB/second) and the number of messages sent per second (records/second).
Kafka producer test script: `$KAFKA_HOME/bin/kafka-producer-perf-test.sh`
Kafka consumer test script: `$KAFKA_HOME/bin/kafka-consumer-perf-test.sh`

Testing Command

Note:
The `ckafka vip:vport` in the following sample commands should be replaced by the actual IP and port assigned for your instance.
Sample command for production testing:

```
bin/kafka-producer-perf-test.sh
--topic test
--num-records 123
--record-size 1000
--producer-props bootstrap.servers= ckafka vip : port
--throughput 20000
```

Sample command for consumption testing:

```
bin/kafka-consumer-perf-test.sh
--topic test
--new-consumer
--fetch-size 10000
--messages 1000
--broker-list bootstrap.servers=ckafka vip : port
```

Suggestions

We recommend you create three or more partitions to increase throughput. This is because there must be at least three CKafka cluster nodes at the backend. If only one partition is created, it will be distributed in a single broker, which will affect CKafka performance.
As messages in each CKafka partition are ordered, the production performance will be affected if there are too many partitions. We recommend you create up to six partitions.
It is necessary to simulate concurrency with multiple clients to ensure the testing effect. We recommend you use multiple test servers as the pressure test clients (producers) and start multiple pressure test programs on each test server to increase concurrency. To avoid the high load of test servers, you are also recommended to start one producer every second rather than all producers simultaneously.

Configuration Guide for Common Parameters in CKafka

Last updated：2024-01-09 15:02:48

Broker Configuration Parameter Description

The following are the configurations of a CKafka Broker for your reference:

```
# Maximum message length in bytes.
message.max.bytes=1000012

# Whether to allow automatic creation of topics. The default value is false. Curren
auto.create.topics.enable=false

# Whether to allow topic deletion by calling the API.
delete.topic.enable=true

# The maximum request length allowed for a Broker is 16 MB.
socket.request.max.bytes=16777216

# Each IP can establish up to 5,000 connections with a Broker.
max.connections.per.ip=5000

# Offset retention period. The default value is 7 days.
offsets.retention.minutes=10080

# Everyone is allowed to access when there is no ACL configuration.
allow.everyone.if.no.acl.found=true

# The log segment size is 1 GB.
log.segment.bytes=1073741824

# The log rolling check interval is 5 minutes. If the retention period is set to le
log.retention.check.interval.ms=300000
```

Note:

For configurations not listed here, see the open-source Kafka default configurations.

Topic Configuration Parameter Description

1. Number of partitions

From the producer's point of view, writes to different partitions are completely in parallel; from the consumer's point of view, the number of concurrencies depends entirely on the number of partitions (if there are more consumers than partitions, there will definitely be idle consumers). It is important to select an appropriate number of partitions to fully play the performance of the CKafka instance.
The number of partitions should be determined based on the throughput of production and consumption, ideally through the following formula:
Note:
Num = max(T/PT, T/CT) = T/min(PT, CT)
Num represents the number of partitions, T the target throughput, PT the maximum production throughput by the producer to a single partition, and CT the maximum consumption throughput by the consumer from a single partition. The number of partitions is equal to T/PT or T/CT, whichever is larger.
In practice, the actual PT is determined by batch size, compression algorithm, acknowledgement mechanism, number of replicas, and so on, while the actual CT is subject to business logic, which varies according to the actual conditions.
We recommend that the number of partitions be greater than or equal to that of consumers to achieve maximum concurrency. For example, if there are 5 consumers, there should be 5 or more partitions.
However, having too many partitions will lower production throughput and increase time consumed by elections and thus need to be avoided. See the following for reference:
A single partition can implement sequential writes of messages.
A single partition can only be consumed by a single consumer process in the same consumer group.
A single consumer process can consume multiple partitions simultaneously, so partition limits the concurrency of consumers.
The more partitions there are, the longer it takes to elect a leader upon failure.
Offset can be down to the partition level. The more partitions there are, the more time the offset query consumes.
The number of partitions can be dynamically increased but not reduced. However, an increase will result in message rebalance.

2. Number of replicas

At present, the number of replicas must be at least 2 to ensure availability. To ensure high reliability, we recommend maintaining at least 3 replicas.
Note:
The number of replicas will affect the production/consumption traffic; for example, if there are 3 replicas, the actual traffic will be 3 times the production traffic.

3. Log retention period

The `log.retention.ms` configuration of a topic is set through the retention period of the instance in the console.

4. Other topic-level configurations

```
# Maximum message length at the topic level.
max.message.bytes=1000012

# Messages in the 0.10.2 version are in the V1 format.
message.format.version=0.10.2-IV0

# Replica not in ISR can be selected as a leader; in this case, availability is hig
unclean.leader.election.enable=true

# Minimum number of replicas for producer requests submitted by ISR. If the number
min.insync.replicas=1
```

Producer Configuration Guide

The following describes common parameter settings for the Producer client. We recommend adjusting them based on your actual business scenarios:

```
# The producer will attempt to bundle and send the messages sent to the same partit
batch.size=16384

# The following describes the 3 ACK mechanisms supported by a Kafka producer:
# -1 or all: the Broker responds to the producer and continues to send the next mes
# 0: the producer continues to send the next message or next batch of messages with
# 1: the producer sends the next message or next batch of messages after it receive

# If users do not configure this, the default value will be 1. Users can customize
acks=1

# Control the maximum time a production request waits in the Broker for replica syn
timeout.ms=30000

# Configure the memory that the producer uses to cache messages to be sent to the B
buffer.memory=33554432

# If messages are produced faster than they are sent by the sender thread to the Br
max.block.ms=60000

# Set the time to send the scheduled message, so that more messages can be sent in
linger.ms=0

# Maximum size of the request packet that the producer can send, which defaults to
max.request.size=1048576
```

```
# Compression format configuration. Currently, version 0.9 and earlier do not suppo
compression.type=[none, snappy, lz4]

# Timeout period for the client to send a request to the Broker, which cannot be sm
request.timeout.ms=30000

# Maximum number of unacknowledged requests that the client can send on each connec
max.in.flight.requests.per.connection=5


# Number of retries upon request error. It is recommended that you set the paramete
retries=0

# Retry interval upon request failure.
retry.backoff.ms=100
```

Consumer Configuration Guide

The following describes common parameter settings for the Consumer client. We recommend adjusting them based on your actual business scenarios:

```
# Whether to sync the offset to the Broker after a message is consumed, so the late
auto.commit.enable=true

# Interval for the automatic submission of offset when auto.commit.enable=true is c
auto.commit.interval.ms=5000

# Mode to initialize the offset when no offset is configured for the Broker (such a
# earliest: reset to the minimum offset in the partition.
# latest: reset to the maximum offset in the partition. This is the default value.
# none: throw an OffsetOutOfRangeException exception without resetting the offset.
auto.offset.reset=latest

# Identify the consumer group to which the consumer belongs.
group.id=""

# Consumer timeout period when the Kafka consumer groups are used. If the Broker do
session.timeout.ms=10000

# Interval at which the consumer sends a heartbeat when the Kafka consumer groups a
heartbeat.interval.ms=3000

# Maximum interval allowed for calling the poll again when the Kafka consumer group
```

```
max.poll.interval.ms=300000

# Minimum data size returned by a fetch request. The default value is 1 B, indicati
fetch.min.bytes=1

# Maximum data size returned by a fetch request. The default value is 50 MB.
fetch.max.bytes=52428800

# Fetch request wait time.
fetch.max.wait.ms=500

# Maximum data size returned by each partition in a fetch request. The default valu
max.partition.fetch.bytes=1048576

# Number of records returned in one poll call.
max.poll.records=500

# Client request timeout period. If no response is received after this time elapses
request.timeout.ms=305000
```

Connecting to Legacy Self-Built Kafka

Last updated：2024-01-09 15:02:48

CKafka is compatible with the Producer and Consumer APIs of Apache Kafka 0.9 and above (currently, directly purchasable versions include v0.10.2, v1.1.1, v2.4.1, v2.8.1, and v3.2.3). To connect to an on-premises Kafka of an earlier version (such as v0.8), partial rewriting of APIs is needed. This document compares the Producer and Consumer APIs of Kafka 0.8 and its earlier versions, and describes how to rewrite the APIs.

Kafka Producer

Overview

In Kafka 0.8.1, the Producer API is rewritten. This Producer client version is officially recommended because it provides better performance and more features. The community will maintain the new version of the Producer API (referred to as the New Producer API).

Comparison between new producer API and old producer API

New Producer API Demo

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:4242");
props.put("acks", "all");
props.put("retries",0);
props.put("batch.size", 16384);
props.put("linger.ms", 1);
props.put("buffer.memory", 33554432);
props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer
props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializ
Producer<String, String> producer = new KafkaProducer<>(props);
producer.send(new ProducerRecord<String, String>("my-topic", Integer.toString(0), I
producer.close();
```

Old Producer API Demo

```
Properties props = new Properties();
props.put("metadata.broker.list", "broker1:9092");
props.put("serializer.class", "kafka.serializer.StringEncoder");
props.put("partitioner.class", "example.producer.SimplePartitioner");
props.put("request.required.acks", "1");
ProducerConfig config = new ProducerConfig(props);
Producer<String, String> producer = new Producer<String, String>(config);
KeyedMessage<String, String> data = new KeyedMessage<String, String>("page_visits",
```

```
producer.send(data);
producer.close();
```

As shown in the previous code, the basic usage of the new and old versions are the same, except for some parameter settings. This means the cost of API partial rewriting is not high.

Compatibility description

Producer API 0.8.x can be connected to CKafka successfully without partial rewriting. We recommend using the New Kafka Producer API.

Kafka Consumer

Overview

The open-source Apache Kafka 0.8 provides two types of the Consumer API:
High Level Consumer API (blocking configuration details)
Simple Consumer API (support for parameter configuration adjustment)
Kafka 0.9.x has introduced the New Consumer API that inherits the features of the two types of the old consumer API (v0.8) and reduces the load on ZooKeeper.
The following describes how to transform the old consumer API (v0.8) to the new consumer API (v0.9).

Comparison between new consumer API and old consumer API

Old consumer API (v0.8)

High Level Consumer API (Demo)
The High Level Consumer API can meet general requirements if you care only about data, except for message offset. This API, built on the consumer group logic, blocks the offset management, and supports Broker fault handling and Consumer load balancing. It allows developers to get started with the Consumer client quickly.
Consider the following when you use the High Level Consumer API:
If the number of consumer threads is greater than the number of partitions, certain consumer threads cannot obtain data.
If the number of partitions is greater than the number of threads, certain threads consume more than one partition.
The changes in partitions and consumers will affect rebalancing.
Low Level Consumer API (Demo)
The Low Level Consumer API is recommended if you need message offset and features like repeated consumption or skip read, or if you want to consume specific partitions and ensure more consumption semantics. But in this case, you need to handle offsets and Broker exceptions.
When using the Low Level Consumer API, you need to:
Track and maintain the offset and control the consumption progress.

Find the leader of partitions for the topic, and deal with partition changes.

New consumer API (v0.9)

Kafka 0.9.x has introduced the New Consumer API that inherits the features of the two types of Old Consumer API while providing consumer coordination (High Level API) and lower-level access to customize consumption policies. The New Consumer also simplifies the consumer client and introduces a central coordinator to solve the herd effect and split-brain problems resulting from the separate connections to ZooKeeper, and to reduce the load on ZooKeeper.

Advantages:

Introduces coordinators

The current version of High Level Consumer has the herd effect and split-brain problems. Placing failure detection and rebalancing logics into a highly available central coordinator solves both problems while greatly reducing the load on the ZooKeeper.

Allows you to assign partitions

To keep certain states of each local partition unchanged, you need to keep partition mappings unchanged. Some other scenarios are designed to associate the Consumer with the region-dependent Broker.

Allows you to manage offsets

You can manage offsets as needed to implement repeated consumption, skipped consumption, and other semantics.

Triggers callbacks after rebalancing based on your specifications

Provides non-blocking Consumer API

Comparison between new consumer API and old consumer API

Type	Version	Automatic Offset Storage	Manual Offset Management	Automatic Exception Handling	Automatic Rebalance Handling	Automatic Leader Search	Pros and Cons
High Level Consumer	Earlier than v0.9	Yes	No	Yes	Yes	Yes	Herd effect and split br
Simple Consumer	Earlier than v0.9	No	Yes	No	No	No	Various exceptions need to be handled
New Consumer	Later than v0.9	Yes	Yes	Yes	Yes	Yes	Mature and recommend for the curre version

Transforming old consumer to new consumer

New Consumer

```
//The main configuration difference is that the ZooKeeper parameters are replaced.
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("group.id", "test");
props.put("enable.auto.commit", "true");
props.put("auto.commit.interval.ms", "1000");
props.put("session.timeout.ms", "30000");
props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserial
props.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeseri
//Compared with old consumers, new consumers are easier to create.
KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
consumer.subscribe(Arrays.asList("foo", "bar"));
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    for (ConsumerRecord<String, String> record : records)
       System.out.printf("offset = %d, key = %s, value = %s", record.offset(), reco
```

Old Consumer （High Level）

```
//Old consumers require ZooKeeper
Properties props = new Properties();
props.put("zookeeper.connect", "localhost:2181");
props.put("group.id", "test");
props.put("auto.commit.enable", "true");
props.put("auto.commit.interval.ms", "1000");
props.put("auto.offset.reset", "smallest");
ConsumerConfig config = new ConsumerConfig(props);
//Require connector creation
ConsumerConnector connector = Consumer.createJavaConsumerConnector(config);
//Create a message stream
Map<String, Integer> topicCountMap = new HashMap<String, Integer>();
topicCountMap.put("foo", 1);
Map<String, List<KafkaStream<byte[], byte[]>>> streams =
 connector.createMessageStreams(topicCountMap);
//Obtain data
KafkaStream<byte[], byte[]> stream = streams.get("foo").get(0);
ConsumerIterator<byte[], byte[]> iterator = stream.iterator();
MessageAndMetadata<byte[], byte[]> msg = null;
while (iterator.hasNext()) {
     msg = iterator.next();
     System.out.println(//
            " group " + props.get("group.id") + //
            ", partition " + msg.partition() + ", " + //
             new String(msg.message()));
}
```

Comparing with the old consumer, the new consumer has simpler coding and uses Kafka addresses instead of ZooKeeper parameters. In addition, the new consumer has parameter settings for interactions with coordinators, in which the default settings are suitable for general use.

Compatibility description

Both CKafka and the new version of Kafka in the open-source community support the rewritten new consumer API, which blocks the interaction between the consumer client and ZooKeeper (ZooKeeper is not exposed to users any longer). The new consumer solves the herd effect and split brain problems resulting from direct interaction with ZooKeeper, and integrates the features of the old consumer, thus making the consumption more reliable.

Suggestions for CKafka Version Selection

Last updated：2024-11-07 15:38:50

This document describes the compatibility of CKafka with open-source Kafka and helps you select a CKafka version that is more suitable for your business according to your business needs.

Overview

Open-Source Kafka has about 20 versions ranging from v0.7.x to v2.8.x. From the perspective of message queue, it can be divided into three stages: v0.x, v1.x, and v2.x. At present, Tencent Cloud provides four corresponding versions for these three community development stages: v0.10, v1.1, v2.4, and v2.8, which basically cover commonly used Kafka versions.
Among them, the two major versions v1.x and v2.x mainly contain optimizations and improvements of Kafka Streams but don't introduce many major features in terms of the message engine (v2.x has great improvements of the transaction feature though). Kafka Streams is greatly improved on v2.x. Therefore, if you need to use these features, please select v2.x at least.

Compatibility Description

CKafka is perfectly compatible with open-source Kafka with full backward compatibility with lower versions. For example, if you build Kafka v0.10 on your own, you can select CKafka v0.10, v1.1.1, or v2.4.1 in the cloud, but if you build Kafka on a higher version, we recommend you not select a lower version (because it is uncertain whether your business uses features of higher versions).
The following describes the compatibility:

CKafka Version	Compatible Community Versions	Compatibility
0.10.2	≤ 0.10.x	100%
1.1.1	≤ 1.1.x	100%
2.4.1	≤ 2.4.x	100%
2.8.1	≤ 2.8.x	100%
3.2.3	≤ 3.2.x	100%

Note for CKafka v2.4.1

When CKafka launched v2.4, the stable branch in the community was v2.4.1. Later, the community launched a development branch v2.4.2. After some repairs were merged, it was positioned as v2.4.2. Then, the community deleted v2.4.2, so there was no v2.4.2 in the community eventually. Therefore, the v2.4.2 previously displayed by CKafka is aligned with the current v2.4.1.

Suggestions for CKafka Version Selection

If you migrate your self-built Kafka to the cloud, we recommend you select the corresponding major version. For example, if your self-built Kafka is on v1.1.0, please select CKafka v1.1.
If the corresponding version cannot be found in the cloud, we recommend you select a higher version. For example, if your self-built Kafka is on v1.0.0, we recommend you use CKafka v1.1.1, and if it is on v0.11.x, we also recommend you use CKafka v1.1.1 (because each version of Broker is backward compatible).

CKafka Data Reliability Description

Last updated：2024-01-09 15:02:47

This document describes the factors that affect the reliability of CKafka from the perspectives of the producer, the server (CKafka), and the consumer, respectively, and provides corresponding solutions.

What should I do if data gets lost on the producer?

Causes of data loss

When the producer sends data to CKafka, the data may get lost due to network jitters, and CKafka will not receive the data. Other possible causes are as follows:
The network load is high or the disk is busy, and the producer does not have a retry mechanism.
The purchased disk capacity is exceeded. For example, if the disk capacity of an instance is 9,000 GB and it is not expanded promptly after being used up, data cannot be written to CKafka.
Sudden or continuously increased peak traffic exceeds the purchased peak throughput. For example, if the peak throughput of the instance is 100 MB/sec, but it is not scaled up promptly after the peak throughput is exceeded for a long period of time, data writes to CKafka will become slower. In this case, if the producer has a queuing timeout mechanism in place, data cannot be written to CKafka.

Solutions

Enable the retry mechanism on the producer for important data.
To avoid data loss caused by improper disk usage, set monitoring and alarm policies as preventive measures when configuring the instance.
When the disk capacity is used up, upgrade the instance timely in the console. Upgrading Ckafka instances of Standard Edition will not interrupt the service. The disk capacity can be expanded separately. You can also shorten the message retention period to reduce disk usage.
To minimize the loss of messages on the producer, you can fine-tune the size of the buffer by using `buffer.memory` and `batch.size` (in bytes). A larger buffer is not necessarily better. When the producer fails for any reason, more data in the buffer means more garbage to be recycled, which slows down data recovery. **Pay close attention to the number of messages produced by the producer and the average message size** (through the rich set of monitoring metrics available in CKafka).
Configure acknowledgment (ACK) for the producer.
When the producer sends data to the leader, it can set the data reliability level by using the `request.required.acks` and `min.insync.replicas` parameters.
When `acks = 1` (default value), the leader in the ISR has successfully received a message sent by the producer, and the next message can be sent. If the leader goes down, the data unsynced to its followers will get lost.

When `acks = 0`, the producer sends the next message without waiting for acknowledgment from the broker. In this case, data transfer efficiency is the highest, but data reliability is the lowest.

Note:

When the producer is configured with `acks = 0`, if the current instance is throttled, in order for the server to provide services normally, the server will actively close the connection to the client.

When `acks = -1` or `acks = all`, the producer needs to wait for the acknowledgment of message receipt from all the followers in the ISR before sending the next message, which ensures the highest reliability.

Even if `acks` is configured as above, there is no guarantee that data will never get lost. For example, when there is only one leader in the ISR (the number of members in the ISR may increase or decrease in certain circumstances, and in some cases, only one leader is left), the value of `acks` will be `1`. Therefore, you also need to configure the `min.insync.replicas` parameter in the CKafka console by enabling the advanced configuration in **Topic Management** > **Edit Topic**. This parameter specifies the minimum number of replicas in the ISR, and its default value is 1. It only takes effect when `acks = -1` or `acks = all`.

Recommended parameter values

These parameter values are for reference only, and the actual values depend on the actual conditions of your business.

Retry mechanism: `message.send.max.retries=3;retry.backoff.ms=10000;`

Guarantee of high reliability: `request.required.acks=-1;min.insync.replicas=2;`

Guarantee of high performance: `request.required.acks=0;`

Reliability + performance: `request.required.acks=1;`

What should I do if data gets lost on the broker (CKafka)?

Causes of data loss

The partition's leader goes down before the followers complete the data backup. Even if a new leader has been selected, data will get lost because it has not been backed up yet.

Open-source Kafka stores data to disks in an async manner. Specifically, data is first stored in PageCache before persistence. If the broker disconnects, restarts, or fails, the data stored in PageCache will get lost because it has not been stored persistently to the disks yet.

Stored data may get lost due to disk failures.

Solutions

Open-source Kafka has multiple replicas that are used to ensure data integrity. Data will get lost only if multiple replicas and brokers fail at the same time, so data reliability is higher than that in the single-replica case. Therefore, CKafka requires at least two replicas for a topic and supports configuring three replicas.

CKafka performs data flushing by configuring more reasonable parameters, such as `log.flush.interval.messages` and `log.flush.interval.ms`.

In CKafka, the disk is specially designed to ensure that data reliability will not be compromised even if the disk is partially damaged.

Recommended parameter values

Whether a replica that is not in sync status can be elected as a leader:

```
unclean.leader.election.enable=false // Disabled
```

What should I do if data gets lost on the consumer?

Causes of data loss

The offset is committed before data is consumed. If the consumer goes down in the process but the offset has been updated, the consumer will miss a data entry, and the consumer group will have to reset the offset in order to retrieve it.

The consumption speed differs significantly from the production speed, but the message retention period is too short; therefore, the message will be deleted upon expiration before it is consumed.

Solutions

Configure the `auto.commit.enable` parameter appropriately. When it is set to `true`, the commit is performed automatically. We recommend that you use the scheduled commit feature to avoid committing offsets frequently.

Monitor the consumer and correctly adjust the data retention period. Monitor the consumption offset and the number of unconsumed messages, and configure an alarm to prevent messages from being deleted upon expiration due to slow consumption.

Troubleshooting data loss

Printing partition and offset information locally for troubleshooting

Below is the code for printing information:

```
Future<RecordMetadata> future = producer.send(new ProducerRecord<>(topic, messageKe
RecordMetadata recordMetadata = future.get();
log.info("partition: {}", recordMetadata.partition());
log.info("offset: {}", recordMetadata.offset());
```

If the partition and offset can be printed out, the currently sent message has been correctly saved on the server. At this time, you can use the message query tool to query the information of the relevant offset.

If the partition and offset information cannot be printed out, the message has not been saved on the server, and the client needs to retry.

Connector
Database Change Subscription
MongoDB Data Subscription

Last updated：2024-09-09 21:38:37

Introduction

The MongoDB Kafka Connector allows monitoring all databases or a single database within a MongoDB instance. It also allows monitoring all collections or a single collection within a database. The connector generates change event messages from Mongo modifications and submits them as a message flow to a Kafka topic. Client applications can consume the messages from the corresponding Kafka topic to process database change events, achieving the goal of monitoring specific databases.
This document summarizes and organizes information from the official MongoDB documentation. For details, see MongoDB Change Events.

Event Format

The following JSON framework illustrates the fields that may appear in all change event messages:

```
{
   _id : { <BSON Object> },
   "operationType" : "<operation>",
   "fullDocument" : { <document> },
   "ns" : {
      "db" : "<database>",
      "coll" : "<collection>"
   },
   "to" : {
      "db" : "<database>",
      "coll" : "<collection>"
   },
   "documentKey" : { "_id" : <value> },
   "updateDescription" : {
      "updatedFields" : { <document> },
      "removedFields" : [ "<field>", ... ],
      "truncatedArrays" : [
         { "field" : <field>, "newSize" : <integer> },
         ...
```

```
        ]
    },
    "clusterTime" : <Timestamp>,
    "txnNumber" : <NumberLong>,
    "lsid" : {
        "id" : <UUID>,
        "uid" : <BinData>
    }
}
```

Some fields may only appear in certain event types. The table below describes the corresponding fields and their meanings.

Field	Type	Description
_id	document	A BSON object used to uniquely identify the even The format of the _id object is as follows: { "_data : <BinData|hex string>}. The type of _data depend on the version of MongoDB. For a complete description of the _data type, see Resume Token
operationType	string	This field indicates the operation types that trigge the change events, including the following 8 types insert, delete, replace, update, drop, rename, dropDatabase, and invalidate.
fullDocument	document	This field indicates the documents affected by the insert, replace, delete, and update operations. Fo insert and replace operations, this field indicates the new document. For delete operations, this fiel is omitted, indicating the document no longer exists. For update operations, this field is shown only if fullDocument is configured as updateLookup.
ns	document	Refers to the namespace, consisting of the database and collection.
ns.db	string	Refers to the database name.
ns.coll	string	Refers to the collection name. For dropDatabase operations, this field is omitted.
to	document	When the operation type is Rename, this field indicates the new collection name. This field is omitted for other operations.

to.db	string	Refers to the name of the new database.
to.coll	string	Refers to the new collection name.
documentKey	document	Refers to the ID of the document modified by the operation.
updateDescription	document	Refers to a document that describes the field modified by the update operation. This field is present only if the event corresponds to an update operation.
updateDescription.updatedFields	document	This field contains the fields modified by the update operation, with the value of the field being the updated value.
updateDescription.removedFields	array	This field contains the fields deleted by the update operation.
updateDescription.truncatedArrays	array	This field records the array truncation performed using one or more of the following pipeline-based updates:$addFields$set$replaceRoot$replaceWi
updateDescription.truncatedArrays.field	string	Indicates the field that was removed.
updateDescription.truncatedArrays.newSize	integer	Refers to the number of elements in the truncated array.
clusterTime	Timestamp	Refers to the oplog timestamp associated with the event. For events involving Multi-Document Transactions, the clusterTime values associated with the event are the same.
txnNumber	NumberLong	Refers to the transaction ID. It appears only when the operation is a Multi-Document Transaction.
lsid	Document	Refers to the session ID associated with the transaction. It appears only when the operation is Multi-Document Transaction.

Event List

Insert Event

```
{
```

```
    _id: { < Resume Token > },
    operationType: 'insert',
    clusterTime: <Timestamp>,
    ns: {
       db: 'engineering',
       coll: 'users'
    },
    documentKey: {
       userName: 'alice123',
       _id: ObjectId("599af247bb69cd8996xxxxxx")
    },
    fullDocument: {
       _id: ObjectId("599af247bb69cd8996xxxxxx"),
       userName: 'alice123',
       name: 'Alice'
    }
}
```

The documentKey field contains both _id and username fields, indicating that the engineering.users collection is sharded, with the shard key being username and _id.

Update Event

```
{
    _id: { < Resume Token > },
    operationType: 'update',
    clusterTime: <Timestamp>,
    ns: {
       db: 'engineering',
       coll: 'users'
    },
    documentKey: {
       _id: ObjectId("58a4eb4a30c75625e0xxxxxx")
    },
    updateDescription: {
       updatedFields: {
          email: 'alice@10gen.com'
       },
       removedFields: ['phoneNumber'],
       truncatedArrays: [ {
          "field" : "vacation_time",
          "newSize" : 36
       } ]
    }
}
```

The following example shows the message content of an `update` event with the `fullDocument : updateLookup` option configured:

```
{
   _id: { < Resume Token > },
   operationType: 'update',
   clusterTime: <Timestamp>,
   ns: {
      db: 'engineering',
      coll: 'users'
   },
   documentKey: {
      _id: ObjectId("58a4eb4a30c75625e0xxxxxx")
   },
   updateDescription: {
      updatedFields: {
         email: 'alice@10gen.com'
      },
      removedFields: ['phoneNumber'],
      truncatedArrays: [ {
         "field" : "vacation_time",
         "newSize" : 36
      } ]
   },
   fullDocument: {
      _id: ObjectId("58a4eb4a30c75625e0xxxxxx"),
      name: 'Alice',
      userName: 'alice123',
      email: 'alice@10gen.com',
      team: 'replication'
   }
}
```

Replace Event

```
{
   _id: { < Resume Token > },
   operationType: 'replace',
   clusterTime: <Timestamp>,
   ns: {
      db: 'engineering',
      coll: 'users'
   },
   documentKey: {
      _id: ObjectId("599af247bb69cd8996xxxxxx")
   },
```

```
    fullDocument: {
        _id: ObjectId("599af247bb69cd8996xxxxxx"),
        userName: 'alice123',
        name: 'Alice'
    }
}
```

The `replace` operation is performed in two steps:

Delete the original document corresponding to the `documentKey` .

Insert a new document with the same `documentkey` .

For a `replace` event, the `fullDocument` field represents the new document inserted.

Delete Event

```
{
    _id: { < Resume Token > },
    operationType: 'delete',
    clusterTime: <Timestamp>,
    ns: {
        db: 'engineering',
        coll: 'users'
    },
    documentKey: {
        _id: ObjectId("599af247bb69cd8996xxxxxx")
    }
}
```

For the delete event message, the `fullDocument` field is omitted.

Drop Event

```
{
    _id: { < Resume Token > },
    operationType: 'drop',
    clusterTime: <Timestamp>,
    ns: {
        db: 'engineering',
        coll: 'users'
    }
}
```

When a collection is deleted, this event is triggered, and it causes the connector subscribing to that collection to generate an invalidate event.

Rename Event

```
{
   _id: { < Resume Token > },
   operationType: 'rename',
   clusterTime: <Timestamp>,
   ns: {
      db: 'engineering',
      coll: 'users'
   },
   to: {
      db: 'engineering',
      coll: 'people'
   }
}
```

When a collection name is modified, this event is triggered, and it causes the connector subscribing to that collection to generate an invalidate event.

Drop Database Event

```
{
   _id: { < Resume Token > },
   operationType: 'dropDatabase',
   clusterTime: <Timestamp>,
   ns: {
      db: 'engineering'
   }
}
```

When a database is deleted, this event is triggered, and it causes the connector subscribing to that collection to generate an invalidate event.
Before a drop database event (dropDatabase) is generated, the system will generate a drop event for each collection in the database.

Invalidate Event

```
{
   _id: { < Resume Token > },
   operationType: 'invalidate',
   clusterTime: <Timestamp>
}
```

For a connector with a subscribed collection, when operations like drop event, rename event, or dropDatabase event that affect the collection are performed, an invalidate event is generated.
For a connector with a subscribed database, a dropDatabase event will generate an invalidate event.

MySQL Data Subscription

Last updated：2024-09-09 21:39:35

Overview

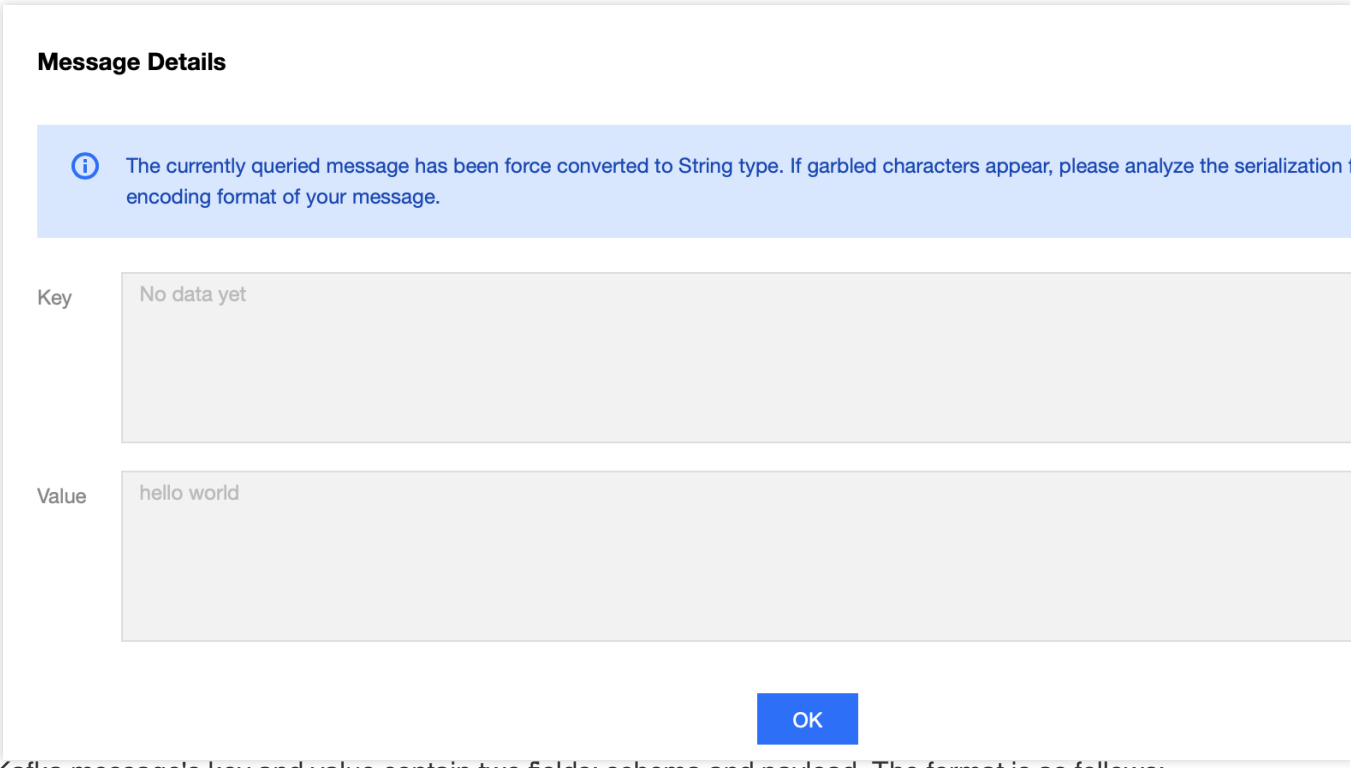
MySQL uses a binary log (binlog) to sequentially record all operations submitted to the database, including modifications to table structures and data within the tables. MySQL uses the binlog for backup or data recovery.
The Debezium MySQL connector generates row-level database change events by reading binlog, including INSERT, UPDATE and DELETE, and sends these events to corresponding Kafka topics. Client applications can process database change events by consuming messages from the corresponding topics to monitor specific databases.
Supported SQL operations for subscription:

Operation Type	Supported SQL Operations
DML	INSERT, UPDATE, and DELETE
DDL	CREATE DATABASE, DROP DATABASE, CREATE TABLE, ALTER TABLE, DROP TABLE, and RENAME TABLE

This document is organized and summarized based on the official Debezium documentation. For details, see Debezium connector for MySQL.

Event Format

The Debezium MySQL connector generates data modifications events for each insert, update, and delete operation. Each event is submitted as a message to the Kafka topic. Each message in the topic contains a key and a value. An example is shown below:



Each Kafka message's key and value contain two fields: schema and payload. The format is as follows:

```
{
 "schema": {
   ...
  },
 "payload": {
   ...
 }
}
```

Key field description:

Item	Field Name	Description
1	schema	The schema field describes the structure of the payload field of the key, i.e., it describes the structure of the primary key of the modified table. If the table does not have a primary key, it describes the structure of its unique key.
2	payload	The structure of the payload field is the same as that described in the first schema and includes the key values of the modified row.

Value field description:

Item	Field Name	Description

1	schema	The schema field describes the structure of the payload field of the value, i.e., it describes the structure of the modified row's fields. This field is usually a nested structure.
2	payload	The structure of the payload field is the same as that described in the second schema and it includes the actual data of the modified row.

Event Message Key

The messages for different types of events all have the same key structure. Below is an example: The key of a change event contains the primary key structure of the modified table and the actual primary key value of the corresponding row.

```
CREATE TABLE customers (
  id INTEGER NOT NULL AUTO_INCREMENT PRIMARY KEY,
  first_name VARCHAR(255) NOT NULL,
  last_name VARCHAR(255) NOT NULL,
  email VARCHAR(255) NOT NULL UNIQUE KEY
) AUTO_INCREMENT=1001;
```

Each event key capturing modifications to the customers table has the same schema. The key of the event message corresponding to this operation is shown as follows (JSON representation):

```
{
 "schema": {
    "type": "struct",
    "name": "mysql-server-1.inventory.customers.Key",
    "optional": false,
    "fields": [
      {
        "field": "id",
        "type": "int32",
        "optional": false
      }
    ]
  },
 "payload": {
    "id": 1001
  }
}
```

Item	Field Name	Description

1	schema	The schema describes the structure in the payload.
2	mysql-server-1.inventory.customers.Key	The naming format of the schema is *connector-name.database-name.table-name*.Key. In this example: The mysql-server-1 is the name of the connector generating the event; the inventory is the name of the corresponding database; the customers is the name of the table.
3	optional	It indicates whether the field is optional.
4	fields	It lists all the fields and their structure contained in the payload, including the field name, field type, and whether it is optional.
5	payload	It contains the primary key of the modified row. In the example, it includes only one primary key value with the field name id: 1001.

DML Events

The previous section introduces the key structure of an event message. The key structures for different types of events are the same. This section lists different event types and describes the value structures for each of these event types.

Create Events

The following example shows the value part of the event message generated by the connector when new data are added to the table:

```
{
  "schema": {
    "type": "struct",
    "fields": [
      {
        "type": "struct",
        "fields": [
          {
            "type": "int32",
            "optional": false,
            "field": "id"
          },
          {
            "type": "string",
            "optional": false,
            "field": "first_name"
          },
          {
            "type": "string",
```

```
            "optional": false,
            "field": "last_name"
          },
          {
            "type": "string",
            "optional": false,
            "field": "email"
          }
        ],
        "optional": true,
        "name": "mysql-server-1.inventory.customers.Value",
        "field": "before"
      },
      {
        "type": "struct",
        "fields": [
          {
            "type": "int32",
            "optional": false,
            "field": "id"
          },
          {
            "type": "string",
            "optional": false,
            "field": "first_name"
          },
          {
            "type": "string",
            "optional": false,
            "field": "last_name"
          },
          {
            "type": "string",
            "optional": false,
            "field": "email"
          }
        ],
        "optional": true,
        "name": "mysql-server-1.inventory.customers.Value",
        "field": "after"
      },
      {
        "type": "struct",
        "fields": [
          {
            "type": "string",
            "optional": false,
```

```
            "field": "version"
          },
          {
            "type": "string",
            "optional": false,
            "field": "connector"
          },
          {
            "type": "string",
            "optional": false,
            "field": "name"
          },
          {
            "type": "int64",
            "optional": false,
            "field": "ts_ms"
          },
          {
            "type": "boolean",
            "optional": true,
            "default": false,
            "field": "snapshot"
          },
          {
            "type": "string",
            "optional": false,
            "field": "db"
          },
          {
            "type": "string",
            "optional": true,
            "field": "table"
          },
          {
            "type": "int64",
            "optional": false,
            "field": "server_id"
          },
          {
            "type": "string",
            "optional": true,
            "field": "gtid"
          },
          {
            "type": "string",
            "optional": false,
            "field": "file"
```

```
          },
          {
            "type": "int64",
            "optional": false,
            "field": "pos"
          },
          {
            "type": "int32",
            "optional": false,
            "field": "row"
          },
          {
            "type": "int64",
            "optional": true,
            "field": "thread"
          },
          {
            "type": "string",
            "optional": true,
            "field": "query"
          }
        ],
        "optional": false,
        "name": "io.debezium.connector.mysql.Source",
        "field": "source"
      },
      {
        "type": "string",
        "optional": false,
        "field": "op"
      },
      {
        "type": "int64",
        "optional": true,
        "field": "ts_ms"
      }
    ],
    "optional": false,
    "name": "mysql-server-1.inventory.customers.Envelope"
  },
  "payload": {
    "op": "c",
    "ts_ms": 1465491411815,
    "before": null,
    "after": {
      "id": 1004,
      "first_name": "Anne",
```

```
      "last_name": "Kretchmar",
      "email": "annek@noanswer.org"
    },
    "source": {
      "version": "1.9.3.Final",
      "connector": "mysql",
      "name": "mysql-server-1",
      "ts_ms": 0,
      "snapshot": false,
      "db": "inventory",
      "table": "customers",
      "server_id": 0,
      "gtid": null,
      "file": "mysql-bin.000003",
      "pos": 154,
      "row": 0,
      "thread": 7,
      "query": "INSERT INTO customers (first_name, last_name, email) VALUES ('Anne'
    }
  }
}
```


Item	Field Name	Description
1	schema	The schema describes the structure in the payload. The field in the schema is an array that represents multiple fields are contained in the payload. Each element in the array is a description of the respective field structure within the payload.
2	field	Each element in the fields includes a field, which indicates the name of the corresponding field in the payload. In the example, it includes before, after, source, etc.
3	type	It indicates the type of field, such as Integer (int) and String (string).
4	mysql-server-1.inventory.customers.Value	It indicates that this field is part of the value information for the customers table in the inventory database generated by the mysql-server-1 connector.
	io.debezium.connector.mysql.Source	This name is bound to a specific connector, and the events generated by the connector all share the same name.
6	payload	It includes the specific modified data in the change event, including the data before (before field) and after (after field) the modification, as well as some metadata information of the connector (source field).

7	op	It indicates the type of modification operation that generates the event. In the example, c indicates an operation that creates a new row. c = create; u = update; d = delete; r = read (only snapshots).
8	source	The source field is a field that describes event metadata. It includes some fields that can be used to compare with other events, such as the order in which events are generated, and whether they belong to the same transaction. This field includes the following metadata information: Debezium version Connector name binlog name where the event was recorded binlog position Row within the event If the event was part of a snapshot Name of the database and table that contain the new row ID of the MySQL thread that created the event (non-snapshot only) MySQL server ID (if available) Timestamp for when the change was made in the database.
9	query	The original SQL statement of the modification operation.

Update Events

The following example shows the value part of the event generated by the update operation:

```
{
  "schema": { ... },
  "payload": {
    "before": {
      "id": 1004,
      "first_name": "Anne",
      "last_name": "Kretchmar",
      "email": "annek@noanswer.org"
    },
    "after": {
      "id": 1004,
      "first_name": "Anne Marie",
      "last_name": "Kretchmar",
      "email": "annek@noanswer.org"
    },
    "source": {
      "version": "1.9.3.Final",
      "name": "mysql-server-1",
      "connector": "mysql",
      "name": "mysql-server-1",
      "ts_ms": 1465581029100,
      "snapshot": false,
      "db": "inventory",
```

```
      "table": "customers",
      "server_id": 223344,
      "gtid": null,
      "file": "mysql-bin.000003",
      "pos": 484,
      "row": 0,
      "thread": 7,
      "query": "UPDATE customers SET first_name='Anne Marie' WHERE id=1004"
    },
    "op": "u",
    "ts_ms": 1465581029523
  }
}
```

The schema field is the same as events in a Create operation, but the payload part is different. In a create event, the before field is null, indicating no original data. In an update event, it includes both the data before and after the update.

Delete Events

The following example shows the value part of the event generated by the delete operation:

```
{
  "schema": { ... },
  "payload": {
    "before": {
      "id": 1004,
      "first_name": "Anne Marie",
      "last_name": "Kretchmar",
      "email": "annek@noanswer.org"
    },
    "after": null,
    "source": {
      "version": "1.9.3.Final",
      "connector": "mysql",
      "name": "mysql-server-1",
      "ts_ms": 1465581902300,
      "snapshot": false,
      "db": "inventory",
      "table": "customers",
      "server_id": 223344,
      "gtid": null,
      "file": "mysql-bin.000003",
      "pos": 805,
      "row": 0,
      "thread": 7,
      "query": "DELETE FROM customers WHERE id=1004"
    },
```

```
    "op": "d",
    "ts_ms": 1465581902461
  }
}
```

The schema field is the same as events in a Create operation, but the payload part is different. In a Delete event, it includes the data before the update, but the data after the update is null, indicating the data has been deleted.

Primary Key Updates

If an operation modifies the primary key of a row in the table, then the connector will generate a **Delete event** to represent the deletion of the row associated with the original primary key, and at the same time generate a **Create event** to represent the insertion of the row associated with the new primary key. The header of each message will be associated with the corresponding key. The official description is as follows:
The `DELETE` event record has `__debezium.newkey` as a message header. The value of this header is the new primary key for the updated row.
The `CREATE` event record has `__debezium.oldkey` as a message header. The value of this header is the previous (old) primary key that the updated row had.

DDL Events

Create Database

The following example shows the value part of the event generated by the **Create Database** operation:

```
{
  "source" : {
    "server" : "dip_source"
  },
  "position" : {
    "ts_sec" : 1655812326,
    "file" : "mysql-bin.000006",
    "pos" : 26063,
    "gtids" : "b24176f2-5409-11ec-80d4-b8599fe5c6ea:1-78",
    "snapshot" : true
  },
  "databaseName" : "dip_test",
  "ddl" : "CREATE DATABASE `dip_test` CHARSET utf8mb4 COLLATE utf8mb4_0900_ai_ci",
  "tableChanges" : [ ]
}
```

The content of position records information such as binlog files and consumption offsets. The ddl field contains the SQL statement that triggers the event.

Drop Database

The following example shows the value part of the event generated by the **Delete Database** operation:

```
{
  "source" : {
    "server" : "dip_source"
  },
  "position" : {
    "ts_sec" : 1655812326,
    "file" : "mysql-bin.000006",
    "pos" : 26063,
    "gtids" : "b24176f2-5409-11ec-80d4-b8599fe5c6ea:1-78",
    "snapshot" : true
  },
  "databaseName" : "dip_test",
  "ddl" : "DROP DATABASE IF EXISTS `dip_test`",
  "tableChanges" : [ ]
}
```

The content of position records information such as binlog files and consumption offsets. The ddl field contains the SQL statement that triggers the event.

Create Table

The following example shows the value part of the event generated by the **create table** operation:

```
{
  "source" : {
    "server" : "dip_source"
  },
  "position" : {
    "ts_sec" : 1655812326,
    "file" : "mysql-bin.000006",
    "pos" : 26063,
    "gtids" : "b24176f2-5409-11ec-80d4-b8599fe5c6ea:1-78",
    "snapshot" : true
  },
  "databaseName" : "dip_test",
  "ddl" : "CREATE TABLE `customers` (\\n  `id` int NOT NULL AUTO_INCREMENT,\\n  `fi
  "tableChanges" : [ {
    "type" : "CREATE",
    "id" : "\\"dip_test\\".\\"customers\\"",
    "table" : {
      "defaultCharsetName" : "utf8",
      "primaryKeyColumnNames" : [ "id" ],
      "columns" : [ {
        "name" : "id",
```

```
        "jdbcType" : 4,
        "typeName" : "INT",
        "typeExpression" : "INT",
        "charsetName" : null,
        "position" : 1,
        "optional" : false,
        "autoIncremented" : true,
        "generated" : true,
        "comment" : null,
        "hasDefaultValue" : false,
        "enumValues" : [ ]
      }, {
        "name" : "first_name",
        "jdbcType" : 12,
        "typeName" : "VARCHAR",
        "typeExpression" : "VARCHAR",
        "charsetName" : "utf8",
        "length" : 255,
        "position" : 2,
        "optional" : false,
        "autoIncremented" : false,
        "generated" : false,
        "comment" : null,
        "hasDefaultValue" : false,
        "enumValues" : [ ]
      }, {
        "name" : "last_name",
        "jdbcType" : 12,
        "typeName" : "VARCHAR",
        "typeExpression" : "VARCHAR",
        "charsetName" : "utf8",
        "length" : 255,
        "position" : 3,
        "optional" : false,
        "autoIncremented" : false,
        "generated" : false,
        "comment" : null,
        "hasDefaultValue" : false,
        "enumValues" : [ ]
      }, {
        "name" : "email",
        "jdbcType" : 12,
        "typeName" : "VARCHAR",
        "typeExpression" : "VARCHAR",
        "charsetName" : "utf8",
        "length" : 255,
        "position" : 4,
```



```
          "optional" : false,
          "autoIncremented" : false,
          "generated" : false,
          "comment" : null,
          "hasDefaultValue" : false,
          "enumValues" : [ ]
        } ]
      },
      "comment" : null
    } ]
}
```

The content of position records information such as binlog files and consumption offsets. The ddl field contains the SQL statement that triggers the event. The columns field records the definition information of the different fields of the new table.

Alter Table

The following example shows the value part of the event generated by the **alter table** operation:

```
{
  "source" : {
    "server" : "1307446078-a123"
  },
  "position" : {
    "transaction_id" : null,
    "ts_sec" : 1655782153,
    "file" : "mysql-bin.000005",
    "pos" : 1218,
    "gtids" : "ddf040ad-7509-11ec-968b-0c42a1eda2e9:1-8",
    "server_id" : 183277
  },
  "databaseName" : "test",
  "ddl" : "ALTER TABLE `user` ADD COLUMN `createtime` datetime NULL DEFAULT CURRENT
  "tableChanges" : [ {
    "type" : "ALTER",
    "id" : "\\"test\\".\\"user\\"",
    "table" : {
      "defaultCharsetName" : "utf8",
      "primaryKeyColumnNames" : [ ],
      "columns" : [ {
        "name" : "name",
        "jdbcType" : 1,
        "typeName" : "CHAR",
        "typeExpression" : "CHAR",
        "charsetName" : "utf8",
        "length" : 20,
```

```
        "position" : 1,
        "optional" : true,
        "autoIncremented" : false,
        "generated" : false,
        "comment" : null,
        "hasDefaultValue" : true,
        "defaultValueExpression" : "",
        "enumValues" : [ ]
      }, {
        "name" : "age",
        "jdbcType" : 4,
        "typeName" : "INT",
        "typeExpression" : "INT",
        "charsetName" : null,
        "position" : 2,
        "optional" : true,
        "autoIncremented" : false,
        "generated" : false,
        "comment" : null,
        "hasDefaultValue" : true,
        "enumValues" : [ ]
      }, {
        "name" : "createtime",
        "jdbcType" : 93,
        "typeName" : "DATETIME",
        "typeExpression" : "DATETIME",
        "charsetName" : null,
        "position" : 3,
        "optional" : true,
        "autoIncremented" : false,
        "generated" : false,
        "comment" : null,
        "hasDefaultValue" : true,
        "defaultValueExpression" : "1970-01-01 00:00:00",
        "enumValues" : [ ]
      } ]
    },
    "comment" : null
  } ]
}
```


The content of position records information such as binlog files and consumption offsets. The ddl field contains the SQL statement that triggers the event. The columns field records the information of the modified fields.

Drop Table

The following example shows the value part of the event generated by the **Drop Table** operation:

```
{
  "source" : {
    "server" : "dip_source"
  },
  "position" : {
    "ts_sec" : 1655812326,
    "file" : "mysql-bin.000006",
    "pos" : 26063,
    "gtids" : "b24176f2-5409-11ec-80d4-b8599fe5c6ea:1-78",
    "snapshot" : true
  },
  "databaseName" : "dip_test",
  "ddl" : "DROP TABLE IF EXISTS `dip_test`.`customers`",
  "tableChanges" : [ ]
}
```

The content of position records information such as binlog files and consumption offsets. The ddl field contains the SQL statement that triggers the event.

Rename Table

The following example shows the value part of the event generated by the **Rename** operation:

```
{
    "schema": {
        "type": "struct",
        "fields": ···,
        "optional": false,
        "name": "io.debezium.connector.mysql.SchemaChangeValue"
    },
    "payload": {
        "source": {
            "version": "1.9.0.Final",
            "connector": "mysql",
            "name": "task-lzpx4pdo",
            "ts_ms": 1656300979748,
            "snapshot": "false",
            "db": "testDB",
            "sequence": null,
            "table": "t_test",
            "server_id": 170993,
            "gtid": "b24176f2-5409-11ec-80d4-b8599fe5c6ea:80",
            "file": "mysql-bin.000006",
            "pos": 26411,
            "row": 0,
            "thread": null,
            "query": null
```

```
        },
        "databaseName": "testDB",
        "schemaName": null,
        "ddl": "rename table test to t_test",
        "tableChanges": [{
            "type": "ALTER",
            "id": "\\"testDB\\".\\"t_test\\"",
            "table": {
                "defaultCharsetName": "utf8",
                "primaryKeyColumnNames": ["id"],
                "columns": [{
                    "name": "id",
                    "jdbcType": -5,
                    "nativeType": null,
                    "typeName": "BIGINT",
                    "typeExpression": "BIGINT",
                    "charsetName": null,
                    "length": 20,
                    "scale": null,
                    "position": 1,
                    "optional": false,
                    "autoIncremented": true,
                    "generated": true,
                    "comment": null
                }, {
                    "name": "name",
                    "jdbcType": 12,
                    "nativeType": null,
                    "typeName": "VARCHAR",
                    "typeExpression": "VARCHAR",
                    "charsetName": "utf8",
                    "length": 20,
                    "scale": null,
                    "position": 2,
                    "optional": true,
                    "autoIncremented": false,
                    "generated": false,
                    "comment": null
                }],
                "comment": null
            }
        }]
    }
}
```


The schema contains the format information of the payload content. Some content is omitted here. In the payload field, source is the metadata information, and the ddl field is the SQL statement that triggers the event. Columns are the

fields of the affected table.

PostgreSQL Data Subscription

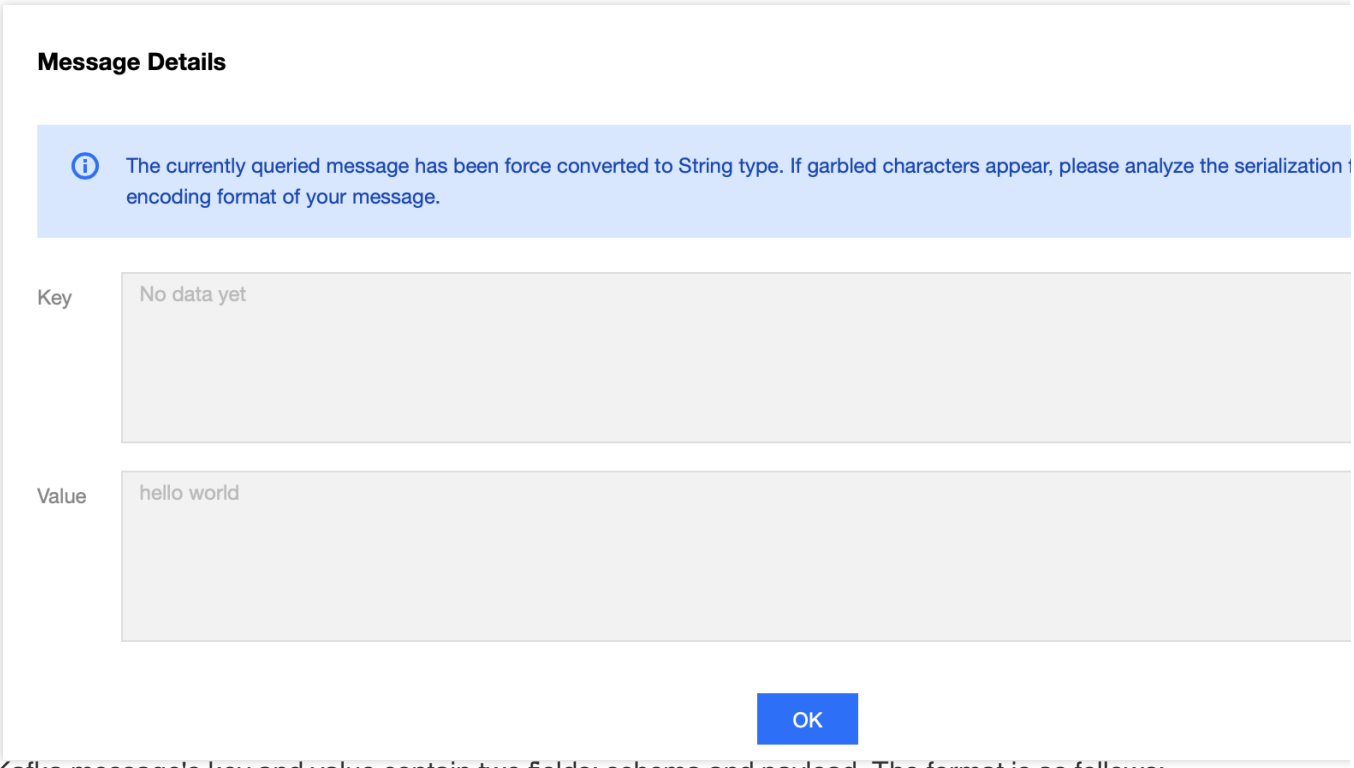
Last updated：2024-09-09 21:40:39

Overview

The Debezium PostgreSQL connector can capture row-level modification operations in the PostgreSQL database and generate corresponding change events. When the Debezium PostgreSQL connector connects to the PostgreSQL server for the first time, it generates a snapshot of all databases. Subsequently, it continuously captures row-level modification operations, including insert, update, and delete, and generates data change events, which are then submitted as messages to the corresponding Kafka topics. Client applications can consume these topic messages to process database change events, enabling monitoring of specific databases.
This document is organized and summarized based on the official Debezium documentation. For details, see Debezium connector for PostgreSQL.

Event Format

The Debezium PostgreSQL connector generates data change events for each row-level insert, update, or delete operation. Each event is submitted as a message to the Kafka Topic. Each message in the Topic contains key and value parts. Here is an example:



Each Kafka message's key and value contain two fields: schema and payload. The format is as follows:

```
{
 "schema": {
   ...
  },
 "payload": {
   ...
 }
}
```

Key field description:

Item	Field Name	Description
1	schema	The schema field describes the structure of the key's payload field, which describes the primary key structure of the modified table. If the table has no primary key, it describes the structure of its unique key.
2	payload	The structure of the payload field is the same as that descried in the schema, and includes the key values of the modified row.

Value field description:

Item	Field Name	Description

1	schema	The schema field describes the structure of the payload field of the value, i.e., it describes the structure of the modified row's fields. This field is usually a nested structure.
2	payload	The structure of the payload field is the same as that described in the schema, and it contains the actual data of the modified row.

Event Message Key

The messages for different types of events all have the same key structure. Below is an example: The key of a change event contains the primary key structure of the modified table and the actual primary key value of the corresponding row.

```
CREATE TABLE customers (
  id SERIAL,
  first_name VARCHAR(255) NOT NULL,
  last_name VARCHAR(255) NOT NULL,
  email VARCHAR(255) NOT NULL,
  PRIMARY KEY(id)
);
```

The key of the event message corresponding to this operation is shown as follows (JSON representation):

```
{
  "schema": {
    "type": "struct",
    "name": "PostgreSQL_server.public.customers.Key",
    "optional": false,
    "fields": [
          {
              "name": "id",
              "index": "0",
              "schema": {
                  "type": "INT32",
                  "optional": "false"
              }
          }
      ]
  },
  "payload": {
      "id": "1"
  },
}
```


Item	Field Name	Description
1	schema	The schema describes the structure in the payload.
2	PostgreSQL_server.inventory.customers.Key	The schema field name follows the format *connector-name.database-name.table-name*.Key. In this example: PostgreSQL_server is the name of the connector generating the event; the inventory is the name of the database; the customers is the name of the table.
3	optional	It indicates whether the field is optional.
4	fields	It lists all the fields and their structure contained in the payload, including the field name, field type, and whether it is optional.
5	payload	It contains the primary key of the modified row. In the example, it includes only one primary key value with the field name id: 1.

Event List

The previous section introduces the structure of an event message key. The key structures for different types of events are the same. This section lists different event types and describes the value structures for each of these event types.

Create Events

Below is a message generated by the Debezium PostgreSQL connector for a database Insert operation:

```
{
    "schema": {
        "type": "struct",
        "fields": [
            {
                "type": "struct",
                "fields": [
                    {
                        "type": "int32",
                        "optional": false,
                        "field": "id"
                    },
                    {
                        "type": "string",
```

```
                    "optional": false,
                    "field": "first_name"
                },
                {
                    "type": "string",
                    "optional": false,
                    "field": "last_name"
                },
                {
                    "type": "string",
                    "optional": false,
                    "field": "email"
                }
            ],
            "optional": true,
            "name": "PostgreSQL_server.inventory.customers.Value",
            "field": "before"
        },
        {
            "type": "struct",
            "fields": [
                {
                    "type": "int32",
                    "optional": false,
                    "field": "id"
                },
                {
                    "type": "string",
                    "optional": false,
                    "field": "first_name"
                },
                {
                    "type": "string",
                    "optional": false,
                    "field": "last_name"
                },
                {
                    "type": "string",
                    "optional": false,
                    "field": "email"
                }
            ],
            "optional": true,
            "name": "PostgreSQL_server.inventory.customers.Value",
            "field": "after"
        },
        {
```

```
                "type": "struct",
                "fields": [
                    {
                        "type": "string",
                        "optional": false,
                        "field": "version"
                    },
                    {
                        "type": "string",
                        "optional": false,
                        "field": "connector"
                    },
                    {
                        "type": "string",
                        "optional": false,
                        "field": "name"
                    },
                    {
                        "type": "int64",
                        "optional": false,
                        "field": "ts_ms"
                    },
                    {
                        "type": "boolean",
                        "optional": true,
                        "default": false,
                        "field": "snapshot"
                    },
                    {
                        "type": "string",
                        "optional": false,
                        "field": "db"
                    },
                    {
                        "type": "string",
                        "optional": false,
                        "field": "schema"
                    },
                    {
                        "type": "string",
                        "optional": false,
                        "field": "table"
                    },
                    {
                        "type": "int64",
                        "optional": true,
                        "field": "txId"
```

```
                    },
                    {
                        "type": "int64",
                        "optional": true,
                        "field": "lsn"
                    },
                    {
                        "type": "int64",
                        "optional": true,
                        "field": "xmin"
                    }
                ],
                "optional": false,
                "name": "io.debezium.connector.postgresql.Source",
                "field": "source"
            },
            {
                "type": "string",
                "optional": false,
                "field": "op"
            },
            {
                "type": "int64",
                "optional": true,
                "field": "ts_ms"
            }
        ],
        "optional": false,
        "name": "PostgreSQL_server.inventory.customers.Envelope"
    },
    "payload": {
        "before": null,
        "after": {
            "id": 1,
            "first_name": "Anne",
            "last_name": "Kretchmar",
            "email": "annek@noanswer.org"
        },
        "source": {
            "version": "1.9.3.Final",
            "connector": "postgresql",
            "name": "PostgreSQL_server",
            "ts_ms": 1559033904863,
            "snapshot": true,
            "db": "postgres",
            "sequence": "[\\"24023119\\",\\"24023128\\"]"
            "schema": "public",
```



```
            "table": "customers",
            "txId": 555,
            "lsn": 24023128,
            "xmin": null
        },
        "op": "c",
        "ts_ms": 1559033904863
    }
}
```

Item	Field Name	Description
1	schema	The schema describes the structure in the payload. The field in the schema is an array that represents multiple fields are contained in the payload. Each element in the array is a description of the respective field structure within the payload.
2	field	Each element in the fields includes a field, which indicates the name of the corresponding field in the payload. In the example, it includes before, after, source, etc.
3	type	It indicates the type of field, such as Integer (int) and String (string).
4	PostgreSQL_server.inventory.customers.Value	It indicates that this field is part of the value information of the customers table in the inventory database generated by the PostgreSQL_server connector.
5	io.debezium.connector.postgresql.Source	This name is bound to a specific connector, and the events generated by the connector all share the same name.
6	payload	It includes the specific modified data in the change event, including the data before (before field) and after (after field) the modification, as well as some metadata information of the connector (source field).
7	op	It indicates the type of modification operation that generates the event. In the example, c indicates an operation that creates a new row. c = create; u = update; d = delete; r = read (only snapshots); t = truncate; m = message

8	source	The source field is a field that describes event metadata. It contains several fields that can be used to compare with other events, such as the order in which events are generated and whether they belong to the same transaction. The source field includes the following metadata: Debezium version; Connector type and name; Database and table that contains the new row; Stringified JSON array of additional offset information (the first value is always the last committed LSN, the second value is the current LSN; either value may be null); Schema name; Whether the event was part of a snapshot; Transaction ID in which the operation was performed; Offset of the operation in the database log; Timestamp for when the change was made in the database.

Update Events

Below is a message generated by the Debezium PostgreSQL connector for a database update operation:

```
{
    "schema": { ... },
    "payload": {
        "before": {
            "id": 1
        },
        "after": {
            "id": 1,
            "first_name": "Anne Marie",
            "last_name": "Kretchmar",
            "email": "annek@noanswer.org"
        },
        "source": {
            "version": "1.9.3.Final",
            "connector": "postgresql",
            "name": "PostgreSQL_server",
            "ts_ms": 1559033904863,
            "snapshot": false,
            "db": "postgres",
            "schema": "public",
            "table": "customers",
            "txId": 556,
            "lsn": 24023128,
            "xmin": null
        },
```

```
        "op": "u",
        "ts_ms": 1465584025523
    }
}
```

The schema field is the same as that in the create event, but the payload part differs. In the create event, the before field is null, indicating there is no original data. In contrast, in the update event, both the before and after fields are present, showing the data before and after the update.

Truncate Events

When a truncate table event occurs, the key of the event message is null. An example of such a message is shown below:

```
{
    "schema": { ... },
    "payload": {
        "source": {
            "version": "1.9.3.Final",
            "connector": "postgresql",
            "name": "PostgreSQL_server",
            "ts_ms": 1559033904863,
            "snapshot": false,
            "db": "postgres",
            "schema": "public",
            "table": "customers",
            "txId": 556,
            "lsn": 46523128,
            "xmin": null
        },
        "op": "t",
        "ts_ms": 1559033904961
    }
}
```

If a TRUNCATE statement affects multiple tables, the connector will generate a separate truncate event message for each affected table.

Message Events

This message type only supports the Postgres 14+ pgoutput plugin. An example of a transaction message event format is shown below:

```
{
    "schema": { ... },
    "payload": {
        "source": {
```

```
            "version": "1.9.3.Final",
            "connector": "postgresql",
            "name": "PostgreSQL_server",
            "ts_ms": 1559033904863,
            "snapshot": false,
            "db": "postgres",
            "schema": "",
            "table": "",
            "txId": 556,
            "lsn": 46523128,
            "xmin": null
        },
        "op": "m",
        "ts_ms": 1559033904961,
        "message": {
            "prefix": "foo",
            "content": "Ymfy"
        }
    }
}
```

An example of a non-transaction message format is shown below:

```
{
    "schema": { ... },
    "payload": {
        "source": {
            "version": "1.9.3.Final",
            "connector": "postgresql",
            "name": "PostgreSQL_server",
            "ts_ms": 1559033904863,
            "snapshot": false,
            "db": "postgres",
            "schema": "",
            "table": "",
            "lsn": 46523128,
            "xmin": null
        },
        "op": "m",
        "ts_ms": 1559033904961
        "message": {
            "prefix": "foo",
            "content": "Ymfy"
    }
}
```

The transaction message event includes a txId field representing the transaction ID. Additionally, the message event contains a message field, which is explained as follows:

Field Name	Description
message	This field contains the metadata of the message: prefix(text) Content (byte array that is encoded based on the binary handling mode setting)

Delete Events

Below is a message generated by the Debezium PostgreSQL connector for a database delete operation:

```
{
    "schema": { ... },
    "payload": {
        "before": {
            "id": 1
        },
        "after": null,
        "source": {
            "version": "1.9.3.Final",
            "connector": "postgresql",
            "name": "PostgreSQL_server",
            "ts_ms": 1559033904863,
            "snapshot": false,
            "db": "postgres",
            "schema": "public",
            "table": "customers",
            "txId": 556,
            "lsn": 46523128,
            "xmin": null
        },
        "op": "d",
        "ts_ms": 1465581902461
    }
}
```

The schema field is the same as that in the create event, but the payload part is different. In a Delete event, it includes the data before the modification, but the data after the update is null, indicating the data has been deleted.

Primary Key Events

If an operation modifies the primary key of a row in the data table, then the connector will generate a **delete event to indicate that the row with the original primary key has been deleted, and a create event is generated** to

represent the insertion of the row with the new primary key. The header of each message will be associated with the corresponding key. The official description is as follows:

The `DELETE` event record has `__debezium.newkey` as a message header. The value of this header is the new primary key for the updated row.

The `CREATE` event record has `__debezium.oldkey` as a message header. The value of this header is the previous (old) primary key that the updated row had.

Official Format Description for MySQL Subscription Messages

Last updated：2024-09-09 21:41:37

Overview

When the CKafka connector is used to subscribe to MySQL change operations, various message formats can be selected. By default, the debezium format is used, and it also provides compatibility with other message formats. This document introduces the message format compatible with **Custom Official Definition Format**.

Official Format I Description

Official Format I currently supports only DML messages. The DDL message format is consistent with the canal format.

Field Name	Field Description
BINLOG_NAME	The binlog file name.
BINLOG_POS	The binlog pos position.
DATABASE	The database name.
EVENT_SERVER_ID	It is temporarily set to null by default.
GLOBAL_ID	If GTID is enabled, this field contains GTID information.
GROUP_ID	It is temporarily set to null by default.
NEW_VALUES	If type = U, this field contains the updated row information in JSON format. If type = D, this field is null. If type = I, this field contains the newly inserted row information in JSON format.
OLD_VALUES	If type = U, this field contains the row information before the update in JSON format. If type = D, this field contains the deleted row information in JSON format. If type = I, this field is null.
TABLE	The table name.
TIME	The log generation time.
TYPE	The log type:

	U: update D: delete I: insert

DDL Format

Create Database

```
{
    "data": null,
    "database": "dip_test",
    "es": 1655812326,
    "id": 0,
    "isDdl": true,
    "mysqlType": null,
    "old": null,
    "pkNames": null,
    "sql": "CREATE DATABASE `dip_test` CHARSET utf8mb4 COLLATE utf8mb4_0900_ai_ci",
    "sqlType": null,
    "table": "",
    "ts": 1655812326,
    "type": "QUERY"
}
```

Drop Database

```
{
    "data": null,
    "database": "dip_test",
    "es": 1655812326,
    "id": 0,
    "isDdl": true,
    "mysqlType": null,
    "old": null,
    "pkNames": null,
    "sql": "DROP DATABASE IF EXISTS `dip_test`",
    "sqlType": null,
    "table": "",
    "ts": 1655812326,
    "type": "QUERY"
}
```

Create Table


```
{
    "data": null,
    "database": "dip_test",
    "es": 1655812326,
    "id": 0,
    "isDdl": true,
    "mysqlType": null,
    "old": null,
    "pkNames": null,
    "sql": "CREATE TABLE `customers` (
  `id` int NOT NULL AUTO_INCREMENT,
  `first_name` varchar(255) NOT NULL,
  `last_name` varchar(255) NOT NULL,
  `email` varchar(255) NOT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `email` (`email`),
  KEY `ix_id` (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=1041 DEFAULT CHARSET=utf8",
    "sqlType": null,
    "table": "customers",
    "ts": 1655812326,
    "type": "CREATE"
}
```


Alter Table


```
{
    "data": null,
    "database": "test",
    "es": 1655782153,
    "id": 0,
    "isDdl": true,
    "mysqlType": null,
    "old": null,
    "pkNames": null,
    "sql": "ALTER TABLE `user` ADD COLUMN `createtime` datetime NULL DEFAULT CURREN
    "sqlType": null,
    "table": "user",
    "ts": 1655782153,
    "type": "ALTER"
}
```

Drop Table

```
{
    "data": null,
    "database": "dip_test",
    "es": 1655812326,
    "id": 0,
    "isDdl": true,
    "mysqlType": null,
    "old": null,
    "pkNames": null,
    "sql": "DROP TABLE IF EXISTS `dip_test`.`customers`",
    "sqlType": null,
    "table": "customers",
    "ts": 1655812326,
    "type": "ERASE"
}
```

Rename Table

```
{
    "data": null,
    "database": "testDB",
    "es": 1656300979748,
    "id": 0,
    "isDdl": true,
    "mysqlType": null,
    "old": null,
    "pkNames": null,
    "sql": "rename table test to t_test",
    "sqlType": null,
    "table": "t_test",
    "ts": 1656300979748,
    "type": "RENAME"
}
```

DML Format

Insert

```
{
    "BINLOG_NAME": "mysql-bin.000003",
    "BINLOG_POS": 154,
```

```
    "DATABASE": "inventory",
    "EVENT_SERVER_ID": null,
    "GLOBAL_ID": null,
    "GROUP_ID": null,
    "NEW_VALUES": {
        "last_name": "Kretchmar",
        "id": "1004",
        "first_name": "Anne",
        "email": "annek@noanswer.org"
    },
    "OLD_VALUES": null,
    "TABLE": "customers",
    "TIME": "19700101080000",
    "TYPE": "I"
}
```

Update

```
{
    "BINLOG_NAME": "mysql-bin.000003",
    "BINLOG_POS": 484,
    "DATABASE": "inventory",
    "EVENT_SERVER_ID": null,
    "GLOBAL_ID": null,
    "GROUP_ID": null,
    "NEW_VALUES": {
        "last_name": "Kretchmar",
        "id": "1004",
        "first_name": "Anne Marie",
        "email": "annek@noanswer.org"
    },
    "OLD_VALUES": {
        "last_name": "Kretchmar",
        "id": "1004",
        "first_name": "Anne",
        "email": "annek@noanswer.org"
    },
    "TABLE": "customers",
    "TIME": "20160611015029",
    "TYPE": "U"
}
```

Delete

```
{
```

```
    "BINLOG_NAME": "mysql-bin.000003",
    "BINLOG_POS": 805,
    "DATABASE": "inventory",
    "EVENT_SERVER_ID": null,
    "GLOBAL_ID": null,
    "GROUP_ID": null,
    "NEW_VALUES": null,
    "OLD_VALUES": {
        "last_name": "Kretchmar",
        "id": "1004",
        "first_name": "Anne Marie",
        "email": "annek@noanswer.org"
    },
    "TABLE": "customers",
    "TIME": "20160611020502",
    "TYPE": "D"
}
```


Canal Format of MySQL Subscription Message

Last updated：2024-01-09 15:02:47

Overview

When using CKafka Connector to subscribe to change operations in MySQL, you can select multiple message formats, and the default format is Debezium. In addition, the system provides compatibility with other message formats. This document describes the message formats compatible with the **official custom format**.

Official Format 1

Official format 1 currently is supported only for DML messages, while the DDL message format is the same as the Canal format.

Field	Description
BINLOG_NAME	Binlog filename
BINLOG_POS	Binlog position
DATABASE	Database name
EVENT_SERVER_ID	It is null currently
GLOBAL_ID	GTID information if GTID is enabled
GROUP_ID	It is null currently
NEW_VALUES	If `type` is `U`, it is the row information after the update in JSON format. If `type` is `D`, it is null. If `type` is `I`, it is the inserted row information in JSON format.
OLD_VALUES	If `type` is `U`, it is the row information before the update in JSON format. If `type` is `D`, it is the deleted row information in JSON format. If `type` is `I`, it is null.
TABLE	Table name
TIME	Log generation time

TYPE	Log type. Valid values: `U` (UPDATE); `D` (DELETE); `I` (INSERT).

DDL Format

create database

```
{
    "data": null,
    "database": "dip_test",
    "es": 1655812326,
    "id": 0,
    "isDdl": true,
    "mysqlType": null,
    "old": null,
    "pkNames": null,
    "sql": "CREATE DATABASE `dip_test` CHARSET utf8mb4 COLLATE utf8mb4_0900_ai_ci",
    "sqlType": null,
    "table": "",
    "ts": 1655812326,
    "type": "QUERY"
}
```

drop database

```
{
    "data": null,
    "database": "dip_test",
    "es": 1655812326,
    "id": 0,
    "isDdl": true,
    "mysqlType": null,
    "old": null,
    "pkNames": null,
    "sql": "DROP DATABASE IF EXISTS `dip_test`",
    "sqlType": null,
    "table": "",
    "ts": 1655812326,
    "type": "QUERY"
}
```

create table


```
{
    "data": null,
    "database": "dip_test",
    "es": 1655812326,
    "id": 0,
    "isDdl": true,
    "mysqlType": null,
    "old": null,
    "pkNames": null,
    "sql": "CREATE TABLE `customers` (
  `id` int NOT NULL AUTO_INCREMENT,
  `first_name` varchar(255) NOT NULL,
  `last_name` varchar(255) NOT NULL,
  `email` varchar(255) NOT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `email` (`email`),
  KEY `ix_id` (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=1041 DEFAULT CHARSET=utf8",
    "sqlType": null,
    "table": "customers",
    "ts": 1655812326,
    "type": "CREATE"
}
```


alter table


```
{
    "data": null,
    "database": "test",
    "es": 1655782153,
    "id": 0,
    "isDdl": true,
    "mysqlType": null,
    "old": null,
    "pkNames": null,
    "sql": "ALTER TABLE `user` ADD COLUMN `createtime` datetime NULL DEFAULT CURREN
    "sqlType": null,
    "table": "user",
    "ts": 1655782153,
    "type": "ALTER"
}
```


drop table

```
{
    "data": null,
    "database": "dip_test",
    "es": 1655812326,
    "id": 0,
    "isDdl": true,
    "mysqlType": null,
    "old": null,
    "pkNames": null,
    "sql": "DROP TABLE IF EXISTS `dip_test`.`customers`",
    "sqlType": null,
    "table": "customers",
    "ts": 1655812326,
    "type": "ERASE"
}
```

rename table

```
{
    "data": null,
    "database": "testDB",
    "es": 1656300979748,
    "id": 0,
    "isDdl": true,
    "mysqlType": null,
    "old": null,
    "pkNames": null,
    "sql": "rename table test to t_test",
    "sqlType": null,
    "table": "t_test",
    "ts": 1656300979748,
    "type": "RENAME"
}
```

DML Format

insert

```
{
    "BINLOG_NAME": "mysql-bin.000003",
    "BINLOG_POS": 154,
    "DATABASE": "inventory",
```

```
    "EVENT_SERVER_ID": null,
    "GLOBAL_ID": null,
    "GROUP_ID": null,
    "NEW_VALUES": {
        "last_name": "Kretchmar",
        "id": "1004",
        "first_name": "Anne",
        "email": "annek@noanswer.org"
    },
    "OLD_VALUES": null,
    "TABLE": "customers",
    "TIME": "19700101080000",
    "TYPE": "I"
}
```

update

```
{
    "BINLOG_NAME": "mysql-bin.000003",
    "BINLOG_POS": 484,
    "DATABASE": "inventory",
    "EVENT_SERVER_ID": null,
    "GLOBAL_ID": null,
    "GROUP_ID": null,
    "NEW_VALUES": {
        "last_name": "Kretchmar",
        "id": "1004",
        "first_name": "Anne Marie",
        "email": "annek@noanswer.org"
    },
    "OLD_VALUES": {
        "last_name": "Kretchmar",
        "id": "1004",
        "first_name": "Anne",
        "email": "annek@noanswer.org"
    },
    "TABLE": "customers",
    "TIME": "20160611015029",
    "TYPE": "U"
}
```

delete

```
{
    "BINLOG_NAME": "mysql-bin.000003",
```

```
    "BINLOG_POS": 805,
    "DATABASE": "inventory",
    "EVENT_SERVER_ID": null,
    "GLOBAL_ID": null,
    "GROUP_ID": null,
    "NEW_VALUES": null,
    "OLD_VALUES": {
        "last_name": "Kretchmar",
        "id": "1004",
        "first_name": "Anne Marie",
        "email": "annek@noanswer.org"
    },
    "TABLE": "customers",
    "TIME": "20160611020502",
    "TYPE": "D"
}
```


User Permission Settings Reference for PostgreSQL Subscription by Connector

Last updated：2024-09-09 21:42:25

Background

When using the CKafka connector to subscribe to a Postgresql database, you need to assign the appropriate permissions to the PostgreSQL users configured in the **Connection Management**. Only users with the corresponding permissions can synchronize messages when accessing the database from an authorized host. This document explains how to set up user permissions and host access permissions.

User Permission Settings

Permissions need to be granted based on the decode plugin being used, as different plugins require different permissions.

User Permissions Settings When the decoderbufs Plugin Is Used

Log in to the PostgreSQL as a superuser, create a role, and grant the role at least `REPLICATION` and `LOGIN` permissions.
Grant permissions:

```
CREATE ROLE userName REPLICATION LOGIN;
```

Note
Superusers have the necessary permissions by default, so if the user is a superuser, you generally don't need to grant the above permissions. However, for security reasons, it is not recommended to use the superuser.

User Permission Settings When the pgoutput Plugin Is Used

Caution
The pgoutput plugin requires the user configured in Connection Management to have superuser permissions.
Step 1: Verify if the user has superuser permissions.

```
// Log in to the postgresql, execute the \\du command to check user permissions.
postgres=# \\du
Role name  | List of roles Attributes
 admin     | Superuser
 postgres  | Superuser, Create role, Create DB, Replication, Bypass RLS
```

```
 slave     | Replication

 // If the configured user does not have superuser permissions, grant the permissio
postgres=# ALTER USER userName WITH SUPERUSER;
```

Step 2: After the user has superuser permissions, follow these steps to grant the necessary permissions.

The connector pgoutput plugin obtains change events by subscribing to publications on the PostgreSQL node. You can either manually create the publication before starting the connector or grant the configured user the permissions to create publications.

Grant the user the following permissions: `Replication` , `CREATE` , and `SELECT` .

```
CREATE ROLE userName REPLICATION LOGIN;

GRANT CREATE ON DATABASE databaseName TO userName;

GRANT SELECT ON TABLE tableName TO userName;
```

The user configured in Connection Management needs to have owner permissions of the subscribed tables. You can grant the user owner permissions as follows:

1. Create a replication group.

```
CREATE ROLE <replication_group>;
```

2. Add the owner of the table to the replication group.

```
GRANT REPLICATION_GROUP TO <original_owner>;
```

3. Add the connector user to the replication group.

```
GRANT REPLICATION_GROUP TO <replication_user>;
```

4. Transfer the owner permissions of the table to the replication group.

```
ALTER TABLE <table_name> OWNER TO REPLICATION_GROUP;
```

Host Access Permission Settings (Required for Self-Built Clusters)

You need to configure the database to allow the host access of the connector. This can be done by setting the corresponding policies in the `pg_hba.conf` file. For detailed information about `pg_hba.conf` , see pg_hba.conf. The configuration file format is as follows:

```
host    databaseName    userName     11.163.0.0/16         md5
host    databaseName    userName    11.163.0.0/16        trust
```

Data Processing
Data Processing Rule Description

Last updated：2024-11-07 11:40:36

Overview

It is usually necessary to perform some cleaning operations on data, such as formatting raw data, parsing specific fields, and converting data format, when CKafka Connector is used to handle data transfer tasks. Developers often need to build their own data cleaning services (ETL).
Logstash is a free and open server-side data processing pipeline that can collect data from multiple sources, transform the data, and send the data to the respective "storage". Logstash has many filter plugins, making it a widely used and powerful data transformation tool.
However, building, configuring, and maintaining Logstash services increases the difficulty of development and Ops. Therefore, CKafka provides data processing services comparable to Logstash. Developers only need to create data processing tasks in the console. The data processing services allow users to edit the corresponding data processing rules, create chained processing tasks, and preview the data processing result.

Feature List

Logstash	Data Processing Services of Connector	Feature
Codec.json	✔	Data parsing (JSON)
Filter.grok	✔	Data parsing (regular expression match)
Filter.mutate.split	✔	Data parsing (character segmentation)
Filter.date	✔	Date format processing
Filter.json	✔	Internal JSON struct parsing
Filter.mutate.convert	✔	Data modification (format conversion)
Filter.mutate.gsub	✔	Data modification (character replacement)
Filter.mutate.strip	✔	Data modification (leading and trailing space

		removal)
Filter.mutate.join	✔	Data modification (field concatenation)
Filter.mutate.rename	✔	Field modification (field renaming)
Filter.mutate.update	✔	Field modification (field update)
Filter.mutate.replace	✔	Field modification (field replacement)
Filter.mutate.add_field	✔	Field modification (field addition)
Filter.mutate.remove_field	✔	Field modification (field deletion)
Filter.mutate.copy	✔	Field modification (field value copying)
Filter.mutate.merge		TODO
Filter.mutate.uppercase		TODO
Filter.mutate.lowercase		TODO

Introduction to Operation Methods

Data Parsing

Logstash processing method:

```
// Codec.json
input {
    file {
        path => "/var/log/nginx/access.log_json""
        codec => "json"
    }
}
// Filter.grok
filter {
    grok {
        match => {
            "message" => "\\s+(?<request_time>\\d+(?:\\.\\d+)?)\\s+"
        }
    }
}
// Filter.mutate.split
filter {
    split {
```

```
            field => "message"
            terminator => "#"
        }
    }
```

Connector processing method:

Select the corresponding data parsing mode, and click the button to preview the result:

Date Format Processing

Logstash processing method:

```
    // Filter.date
    filter {
        date {
            match => ["client_time", "yyyy-MM-dd HH:mm:ss"]
        }
    }
```

Connector processing method:

1.1 Assign a value to a field by presetting the current system time:

1.2 Process the data by using the **Process value** feature:

Internal JSON Struct Parsing

Logstash processing method:

```
    // Filter.json
    filter {
        json {
            source => "message"
            target => "jsoncontent"
        }
    }
```

Connector processing method:

Select the Map operation for a specific field to parse the field into a JSON object:

Data Modification

Logstash processing method:

```
    // Filter.mutate.convert
    filter {
        mutate {
            convert => ["request_time", "float"]
```

```
        }
    }
    // Filter.mutate.gsub
    filter {
        mutate {
            gsub => ["urlparams", ",", "_"]
        }
    }

    // Filter.mutate.strip
    filter {
        mutate {
            strip => ["field1", "field2"]
        }
    }

    // Filter.mutate.join
    filter {
        mutate {
           join => { "fieldname" => "," }
         }
    }
```

Connector processing method:

Select the corresponding **value processing** mode to set the rule:

1.1 Select the data format to change the data format of the corresponding field with one click:

1.2 Use JSONPath syntax to concatenate elements. For example, use the `$.concat($.data.Response.SubnetSet[0].VpcId,&quot;#&quot;,$.data.Response.SubnetSet[0].SubnetId,&quot;#&quot;,$.data.Response.SubnetSet[0].CidrBlock))` syntax to concatenate VPC and subnet attributes, which can be separated with the `#` character.

1.3 The result is as follows:

Field Modification

Logstash processing method:

```
    // Filter.mutate.rename
    filter {
        mutate {
            rename => ["syslog_host", "host"]
        }
    }
    // Filter.mutate.update
```

```
    filter {
        mutate {
            update => { "sample" => "My new message" }
         }
    }
    // Filter.mutate.replace
    filter {
        mutate {
            replace => { "message" => "%{source_host}: My new message" }
        }
    }
    // Filter.mutate.add_field
    filter {
        mutate {
            split => { "hostname" => "." }
            add_field => { "shortHostname" => "%{[hostname][0]}" }
        }
    }
    // Filter.mutate.remove_field
    filter {
        mutate {
            remove_field => ["field_name"]
        }
     }
    // Filter.mutate.copy
    filter {
        mutate {
             copy => { "source_field" => "dest_field" }
        }
    }
```

Connector processing method:

Examples

Example 1: Multi-Level Field Parsing

Input message:

```
{
    "@timestamp": "2022-02-26T22:25:33.210Z",
    "beat": {
        "hostname": "test-server",
        "ip": "6.6.6.6",
        "version": "5.6.9"
    },
```

```
    "input_type": "log",
    "message": "{\\"userId\\":888,\\"userName\\":\\"testUser\\"}",
    "offset": 3030131,
}
```

Output message :

```
{
    "@timestamp": "2022-02-26T22:25:33.210Z",
    "input_type": "log",
    "hostname": "test-server",
    "ip": "6.6.6.6",
    "userId": 888,
    "userName": "testUser"
}
```

Connector configuration:

1.1 The configuration of processing chain 1 is as follows:

1.2 The result of processing chain 1 is as follows:

```
{
    "@timestamp": "2022-02-26T22:25:33.210Z",
    "input_type": "log",
    "message": "{\\"userId\\":888,\\"userName\\":\\"testUser\\"}",
    "hostname": "test-server",
    "ip": "6.6.6.6"
}
```

1.3 The configuration of processing chain 2 is as follows:

1.4 The result of processing chain 2 is as follows:

```
{
    "@timestamp": "2022-02-26T22:25:33.210Z",
    "input_type": "log",
    "hostname": "test-server",
    "ip": "6.6.6.6",
    "userId": 888,
    "userName": "testUser"
}
```

Example 2: Non-JSON Data Parsing

Input message:

```
region=Shanghai$area=a1$server=6.6.6.6$user=testUser$timeStamp=2022-02-26T22:25:33.
```

Output message:

```
{
    "region": "Shanghai",
    "area": "a1",
    "server": "6.6.6.6",
    "user": "testUser",
    "timeStamp": "2022-02-27 06:25:33",
    "processTimeStamp": "2022-06-27 11:14:49"
}
```

Connector configuration:

1.1 Use delimiter $ to parse the original message:

1.2 The preliminary parsing result is as follows:

```
{
  "0": "region=Shanghai",
  "1": "area=a1",
  "2": "server=6.6.6.6",
  "3": "user=testUser",
  "4": "timeStamp=2022-02-26T22:25:33.210Z"
}
```

1.3 Use delimiter **=** for further parsing:

1.4 The parsing result is as follows:

```
{
  "0": "region=Shanghai",
  "1": "area=a1",
  "2": "server=6.6.6.6",
  "3": "user=testUser",
  "4": "timeStamp=2022-02-26T22:25:33.210Z",
  "0.region": "Shanghai",
  "1.area": "a1",
  "2.server": "6.6.6.6",
  "3.user": "testUser",
  "4.timeStamp": "2022-02-26T22:25:33.210Z"
}
```

1.5 Edit and delete fields, adjust the timestamp format, and add the field of the current system time:

The final result is as follows:

```
{
    "region": "Shanghai",
    "area": "a1",
    "server": "6.6.6.6",
```

```
    "user": "testUser",
    "timeStamp": "2022-02-27 06:25:33",
    "processTimeStamp": "2022-06-27 11:14:49"
}
```

Regular Expression Extraction

Last updated：2024-11-07 11:40:11

The data processing feature of CKafka Connector provides the capability to extract message content based on regular expressions. Regular expression extraction uses the open-source regular expression package re2.
Java's standard regular expression package `java.util.regex` and other widely used regular expression packages, such as PCRE, Perlre, and Python(re), use the backtracking policy. That is, when two options `a|b` are available for a pattern, the engine will first try to match `a` . If the match fails, it will reset the input stream and try to match `b` .
If the matching pattern is deeply nested, the policy requires exponential nested parsing of the input data. If the input string is very long, the matching time can be infinitely long
In contrast, the RE2J algorithm uses a nondeterministic finite automaton (NFA) to check all matches in a single parse of the input data, achieving regular expression matching in linear time.
Regular expression extraction of data processing applies to the extraction of specific fields from messages of long array types. Some common extraction patterns are described below.

Example 1: Extracting the Phone Number Field

Input message:

```
{"message":
    [
        {"email":123456@qq.com,"phoneNumber":"13890000000","IDNumber":"130423199301
        {"email":123456789@163.com,"phoneNumber":"15920000000","IDNumber":"61063019
        {"email":usr333@gmail.com,"phoneNumber":"18830000000","IDNumber":"420602198
    ]
}
```

Output message:

```
{
    "0": "\\"phoneNumber\\":\\"13890000000\\"",
    "1": "\\"phoneNumber\\":\\"15920000000\\"",
    "2": "\\"phoneNumber\\":\\"18830000000\\""
}
```

The regular expression used is:

```
"phoneNumber":"(13[0-9]|14[5|7]|15[0|1|2|3|5|6|7|8|9]|18[0|1|2|3|5|6|7|8|9])\\d{8}"
```

Example 2: Extracting the Email Field

Input message:

```
{"message":
    [
        {"email":123456@qq.com,"phoneNumber":"13890000000","IDNumber":"130423199301
        {"email":123456789@163.com,"phoneNumber":"15920000000","IDNumber":"61063019
        {"email":usr333@gmail.com,"phoneNumber":"18830000000","IDNumber":"420602198
    ]
}
```

Output message:

```
{
    "0": "\\"email\\":\\"123456@qq.com\\"",
    "1": "\\"email\\":\\"123456789@163.com\\"",
    "2": "\\"email\\":\\"usr333@gmail.com\\""
}
```

The regular expression used is:

```
"email":"\\w+([-+.]\\w+)*@\\w+([-.]\\w+)*\\.\\w+([-.]\\w+)*"
```

Example 3: Extracting the ID Number Field

Input message:

```
{
    "@timestamp": "2022-02-26T22:25:33.210Z",
    "input_type": "log",
    "operation": "INSERT",
    "operator": "admin",
    "message": "{\\"email\\":\\"123456@qq.com\\",\\"phoneNumber\\":\\"13890000000\\
}
```

Output message. Retain other fields and extract N IDNumber fields from the message separately:

```
{
    "@timestamp": "2022-02-26T22:25:33.210Z",
    "input_type": "log",
    "operation": "INSERT",
    "operator": "admin",
```

```
    "message.0": "130423199301067425",
    "message.1": "610630199109235723",
    "message.2": "42060219880213301X"
}
```

The used regular expression is:

```
[1-9]\\d{5}(18|19|20)\\d{2}((0[1-9])|(1[0-2]))(([0-2][1-9])|10|20|30|31)\\d{3}[0-9X
```

Multiple processing chains are used, and the result of the first processing chain is as follows:

The message field needs to be further processed, and the result of the second processing chain is as follows:

Processing result:

```
{
    "@timestamp": "2022-02-26T22:25:33.210Z",
    "input_type": "log",
    "operation": "INSERT",
    "operator": "admin",
    "message.0": "130423199301067425",
    "message.1": "610630199109235723",
    "message.2": "42060219880213301X"
}
```

The required N IDNumber fields are extracted, the original message field is deleted, and other fields such as operation are retained.

JSONPath Description

Last updated：2024-11-07 11:39:19

Overview

JSON is currently one of the most commonly used format protocols for Internet information transmission. Data processing mainly focus on parsing and handling JSON data.
JSONPath is a message query syntax specification designed for the JSON format. For data processing, simple JSONPath syntax can be used to quickly retrieve the value of a member in a complex nested JSON struct, and extension functions in the **JayWay** library can also be used to aggregate or operate on a certain type of member fields.

Basic Features

Basic Syntax

`$` is a root node operator and represents the root node of the current JSON struct.
`.<childName>` is a dot operator, and `['<childName>']` is a bracket operator. They represent the selected child member named `childName` of the current object.
`..` is a recursive operator and indicates **recursively** obtaining all child members of the current object.
`[<index>]` is a selection operator and indicates obtaining the No. `index` child member of the current iterable object.

Obtaining Specific Member Variable of Nested JSON Struct

The following figure shows the structure of container standard output logs collected by TKE:

```
{
  "@timestamp": 1648803500.63659,
  "@filepath": "/var/log/tke-log-agent/test7/xxxxxxxx-adfe-4617-8cf3-9997aea90ded/c
  "log": "15:00:00.000[4349811564226374227] [http-nio-8081-exec-64] INFO  com.qclou
  "kubernetes": {
    "pod_name": "tke-es-xxxxxxxxxx-n29jr",
    "namespace_name": "default",
    "pod_id": "xxxxxxxx-adfe-4617-8cf3-9997aea90ded",
    "labels": {
      "k8s-app": "tke-es",
      "pod-template-hash": "xxxx95d557",
      "qcloud-app": "tke-es"
    },
```

```
    "annotations": {
      "qcloud-redeploy-timestamp": "1648016531476",
      "tke.cloud.tencent.com/networks-status": "[{\\n    \\"name\\": \\"tke-bridge\
    },
    "host": "10.0.xx.xx",
    "container_name": "nginx",
    "docker_id": "xxxxxxxx49626ef42d5615a636aae74d6380996043cf6f6560d8131f21a4d8ba"
    "container_hash": "nginx@sha256:xxxxxxxx7b29b585ed1aee166a17fad63d344bc973bc638
    "container_image": "nginx"
  }
}
```

When users need to obtain the current Pod name, that is, the `qcloud-app` member field, they can use the `$.kubernetes.labels.qcloud-app` or `$.['kubernetes'].['labels'].['qcloud-app']` JSONPath syntax.

The running result is as follows. The figure shows that the corresponding logs have been successfully read using JSONPath:

Note

When JSONPath is used to handle parameters, only the square bracket operator can be used if a JSON variable name contains special characters such as `.` .

For example, for a JSON struct like `{"key1.key2":"value1"}` , `$.['key1.key2']` must be used to obtain the corresponding member fields.

Advanced Features

Advanced Syntax

`*` is a wildcard operator and indicates obtaining **all** child objects of the current object.

`*~` is a built-in function and indicates obtaining the names of all child objects of the current iterable object.

`min()` is a built-in function and indicates obtaining the minimum value of child objects of the current iterable object.

`max()` is a built-in function and indicates obtaining the maximum value of child objects of the current tterable object.

`sum()` is a built-in function and indicates obtaining the sum of child objects of the current iterable object.

`concat()` is a built-in function that concatenates multiple objects into a string.

Aggregating Data of Specific Fields

When there is an object list in a JSON struct, the list length is usually variable. Take the request response log in the figure below as an example:

```
{
  "data": {
    "Response": {
      "Result": {
        "Routers": [
          {
            "AccessType": 0,
            "RouteId": 81111,
            "VpcId": "vpc-xxxxxxxx",
            "VipType": 3,
            "VipList": [
              {
                "Vip": "10.0.0.189",
                "Vport": "9xxx"
              }
            ]
          },
          {
            "AccessType": 0,
            "RouteId": 81112,
            "VpcId": "vpc-r5sbavzp",
            "VipType": 3,
            "VipList": [
              {
                "Vip": "10.0.0.248",
                "Vport": "9xxx"
              }
            ]
          },
          {
            "AccessType": 0,
            "RouteId": 81113,
            "VpcId": "vpc-xxxxxxxx",
            "VipType": 3,
            "VipList": [
              {
                "Vip": "10.0.0.210",
                "Vport": "9xxx"
              }
            ]
          }
        ]
      },
      "RequestId": "20e74750-ca40-403d-9ea9-d3f63b5415d2"
    }
  },
  "code": 0
```



```
}
```

When you need to aggregate the member attributes of a variable-length list, you cannot use the processing chain for aggregation. Instead, you can use the JSONPath syntax `*` to match all elements in the list.

For example, if you want to obtain all the Vip elements in VipList, you can use the JSONPath syntax `$.data.Response.Result.Routers[*].VipList[0].Vip`.

The running result is shown below. It shows that all Vip elements in the struct are successfully obtained.

Concatenating and Modify Struct Members

In some scenes, multiple objects in a JSON struct need to be concatenated during data processing to facilitate transfer to the downstream for further operations. The following syntax can be used:

```
{
  "data": {
    "Response": {
      "SubnetSet": [
        {
          "VpcId": "vpc-xxxxxxxx",
          "SubnetId": "subnet-xxxxxxxx",
          "SubnetName": "ckafka_cloud_subnet-1",
          "CidrBlock": "10.0.0.0/19",
          "Ipv6CidrBlock": "",
          "IsDefault": false,
          "IsRemoteVpcSnat": false,
          "EnableBroadcast": false,
          "Zone": "ap-changsha-ec-1",
          "RouteTableId": "rtb-xxxxxxxx",
          "NetworkAclId": "",
          "TotalIpAddressCount": 8189,
          "AvailableIpAddressCount": 8033,
          "CreatedTime": "2021-01-25 17:31:00",
          "TagSet": [],
          "CdcId": "",
          "IsCdcSubnet": 0,
          "LocalZone": false,
          "IsShare": false
        }
      ],
      "TotalCount": 1,
      "RequestId": "705c4955-0cd9-48b2-9132-79eadae2e3e6"
    }
  },
  "code": 0
```

```
}
```

If the downstream has no computing capability, and the VPC and subnet attributes need to be concatenated during data processing, the JSONPath function `concat()` can be used to concatenate multiple fields and modify the output string.

For example, the syntax `$.concat($.data.Response.SubnetSet[0].VpcId,"#",$.data.Response.SubnetSet[0].SubnetId,"#",$.data.Response.SubnetSet[0].CidrBlock))` can be used to concatenate the VPC and subnet attributes, which can be separated with the character `#` .

The running result is shown below. It shows that the information on concatenated VPC resources are successfully obtained:

Self-Built Cluster Connection Instructions (CLB Method)

Last updated：2024-09-09 21:45:28

Overview

When self-built services on Cloud Virtual Machine (CVM) are connected through the CKafka connector, it is necessary to follow the cross-VPC resource access solution standard set by the Tencent Cloud Network Team. You need to first mount your self-built services onto the Cloud Load Balancer (CLB) to achieve cross-VPC resource access to resources.
That is, when you build your MySQL or Mongo services on CVM and use them as a data source to connect the CKafka connector, you need to connect them by mounting the CLB. This document describes the related operation process.

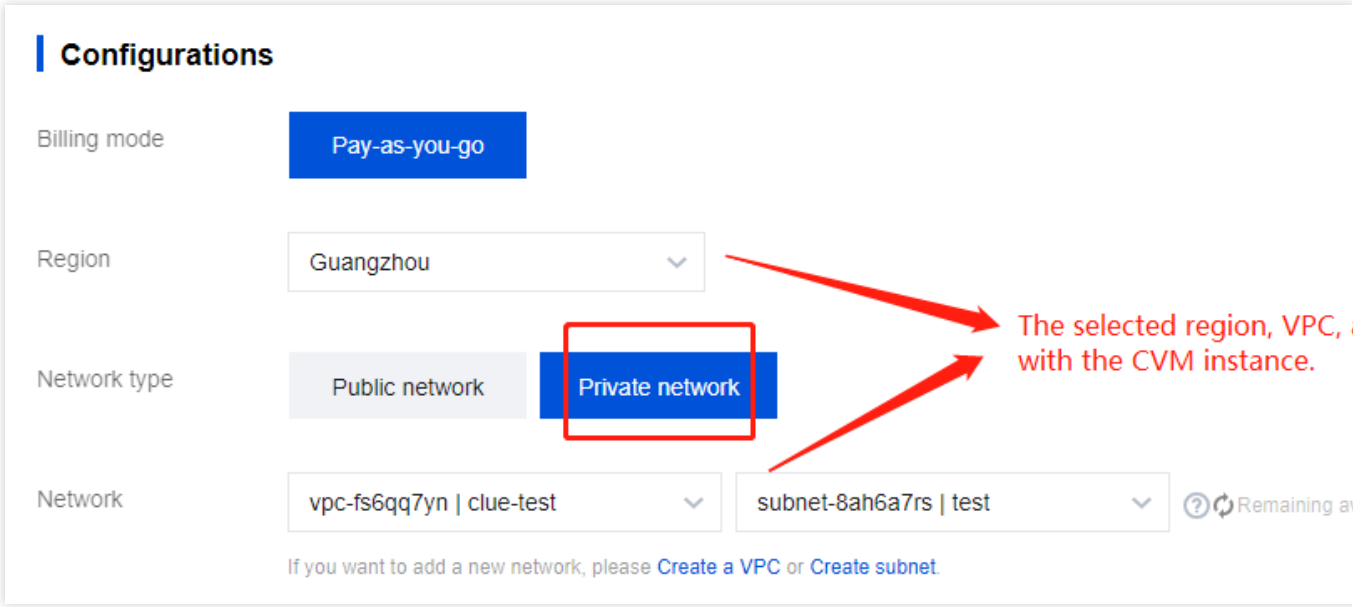
Operation Process

Step 1: Purchasing a CLB Instance (Optional)

If the customer's account already has a private network CLB instance, it can be reused to save costs. Alternatively, a separate CLB instance can be created to provide services. Below is the operation process for creating a new private network CLB.

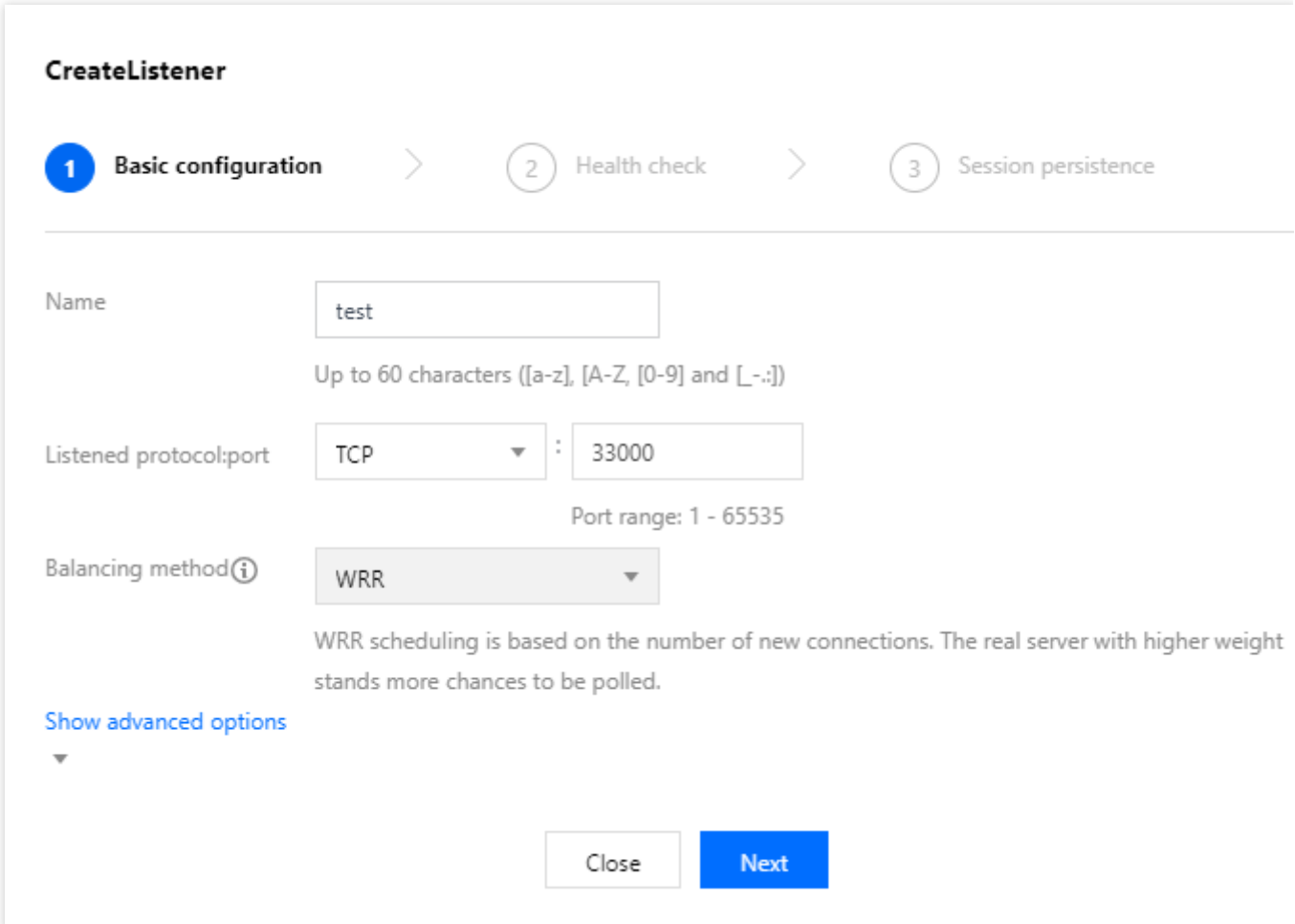
Note

When a CLB instance is purchased, ensure it is in the same region and VPC as that of the CVM cluster. This ensures the CLB can be properly mounted to the CVM instance. Also, choose to purchase a private network CLB instance.



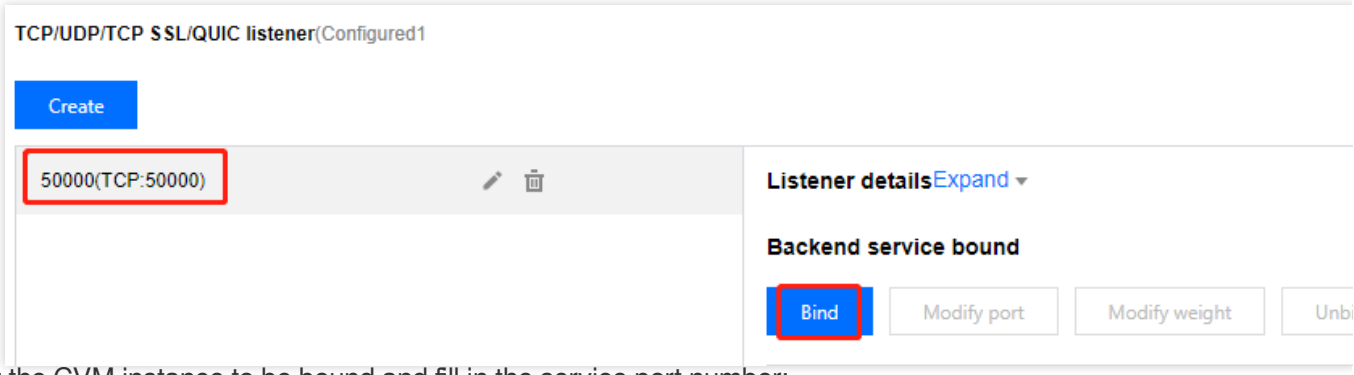
Step 2: Configuring the CLB Instance

1. Create a TCP listener: Enter the **CLB Console** > **Instance Management** page and find the instance to be configured. On the instance details page, click **Listener Management**, and then click Create **TCP Listener**.

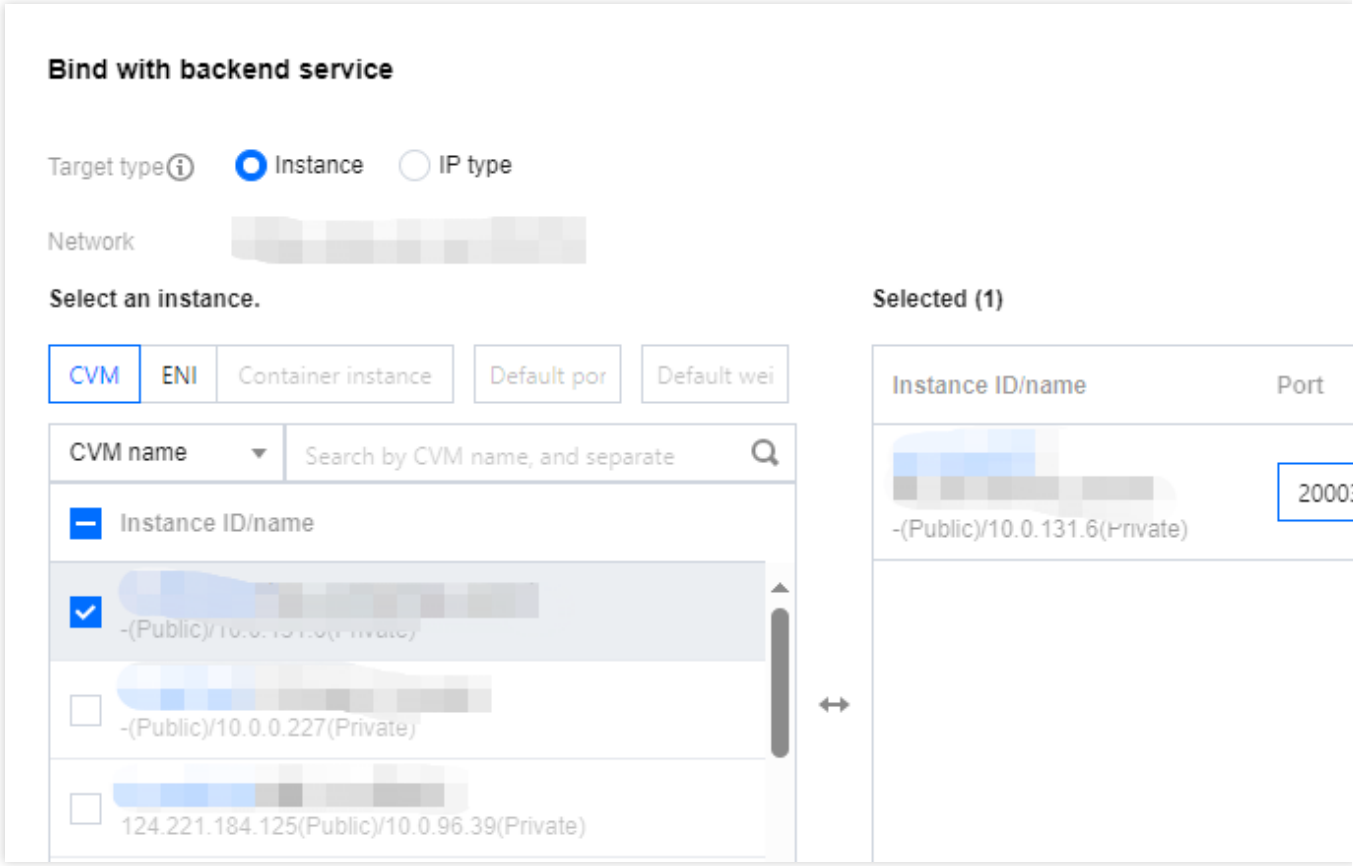


The port configured here is the port used to access the CLB. Health checks and session persistence can be configured based on actual needs.

2. Bind self-built services: After successful creation of the listener, click the corresponding listener, and then click Bind on the right side to bind the CVM instance.



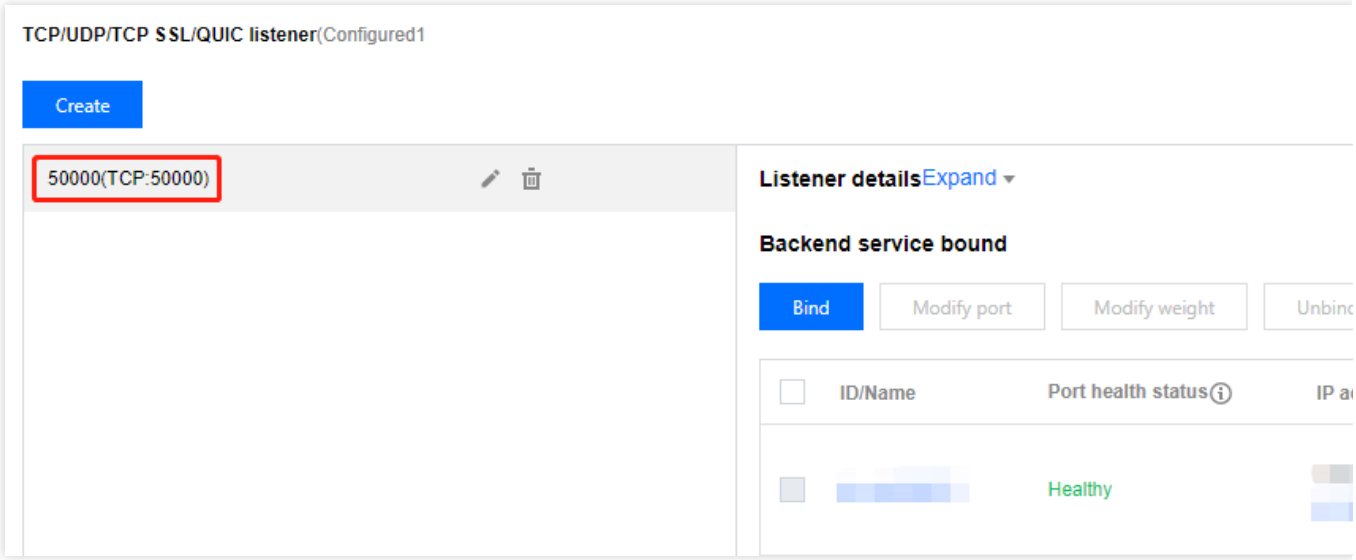
Select the CVM instance to be bound and fill in the service port number:



Note:

For self-built MySQL clusters, it is recommended to bind only **one** CVM instance (either a primary node or a secondary node).This is because there may be a delay in binlog synchronization between the MySQL primary and secondary databases, which can cause binlog read errors when the connector's requests are forwarded to different MySQL services. Therefore, it is recommended to bind only a single MySQL service in self-built clusters.

3. View service health status: After the service is created, you can view the corresponding services and their health status.

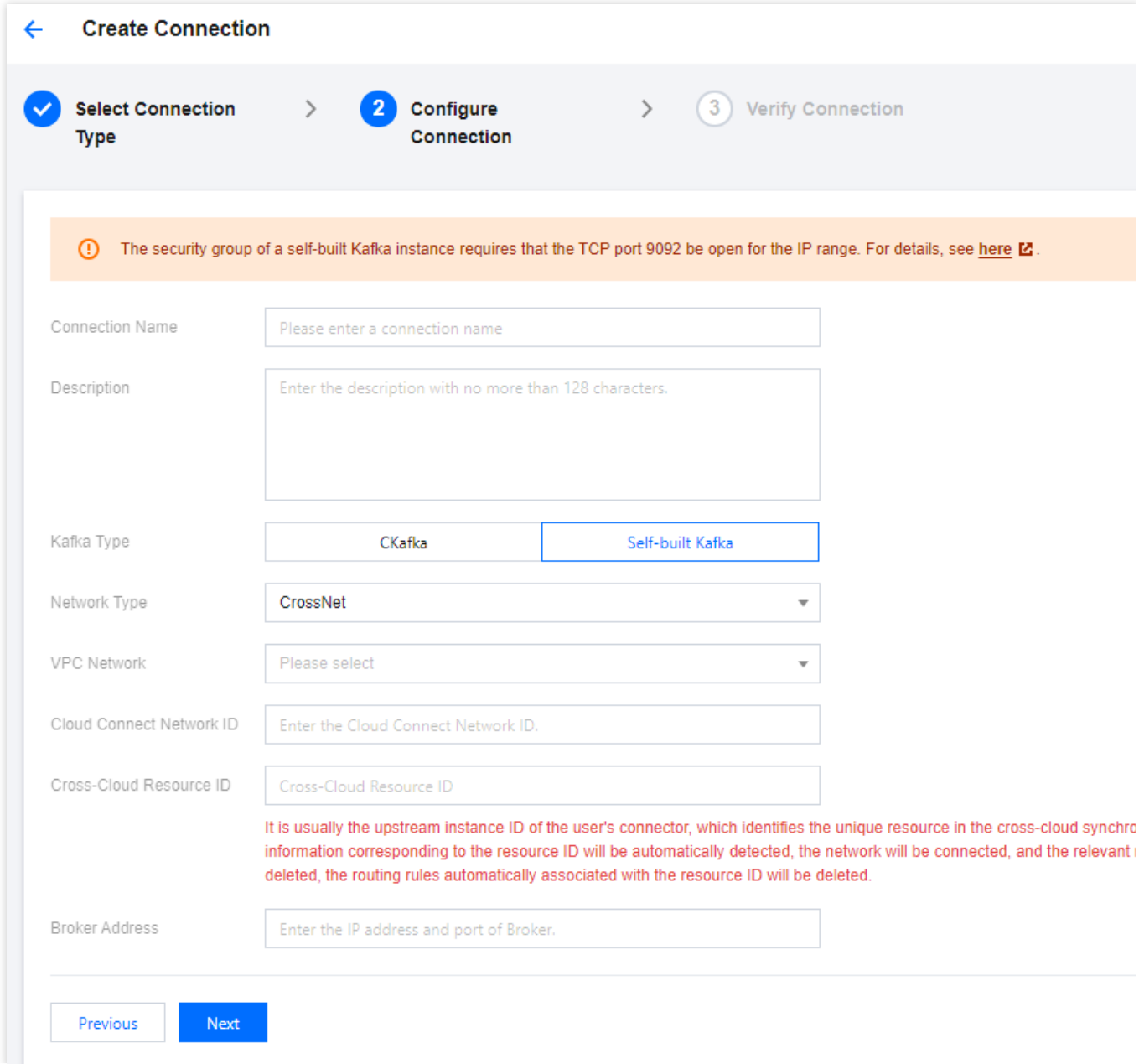


Step 3: Creating a Connection

Enter the CKafka console, click **Connector** > **Connection List**, and then click **New Connection**.

Note

Select the CLB instance that is mounted to the CVM service, the port should be the corresponding CLB listener port, and the username and password should be the ones corresponding to the service.



Authorization Instructions for Access to CLS and COS Services Through Connectors

Last updated：2024-09-09 21:46:31

Overview

When the CKafka connector is used to access services like CLS and COS, users need to grant the connector permissions to access these services under their accounts. If the CKafka sub-account has the CAM policy permissions (QcloudCamRoleFullAccess), select **Role Authorization** when a CKafka task is created, and the connector will automatically complete the authorization for you. Otherwise, a user with AdministratorAccess need to grant the necessary permissions before a connector task is created through the sub-account.

List of Services Requiring Authorization

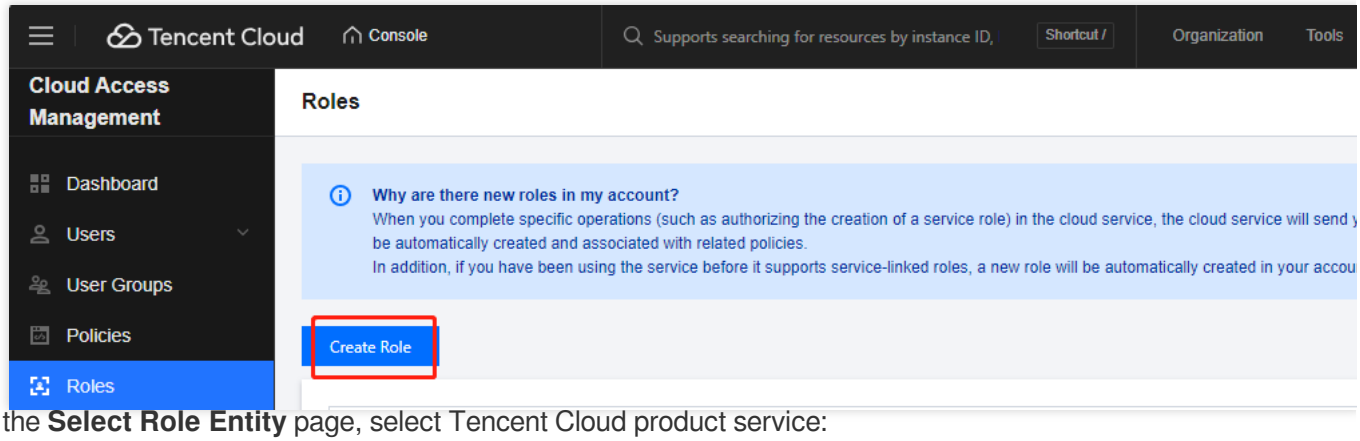
Service Requiring Authorization	Associated Role	Required Policy Permission
Cloud Log Service (CLS)	Datahub_QcsRole	QcloudCLSFullAccess
Cloud Object Storage (COS)	Datahub_QcsRole	QcloudCOSFullAccess

Authorization Steps

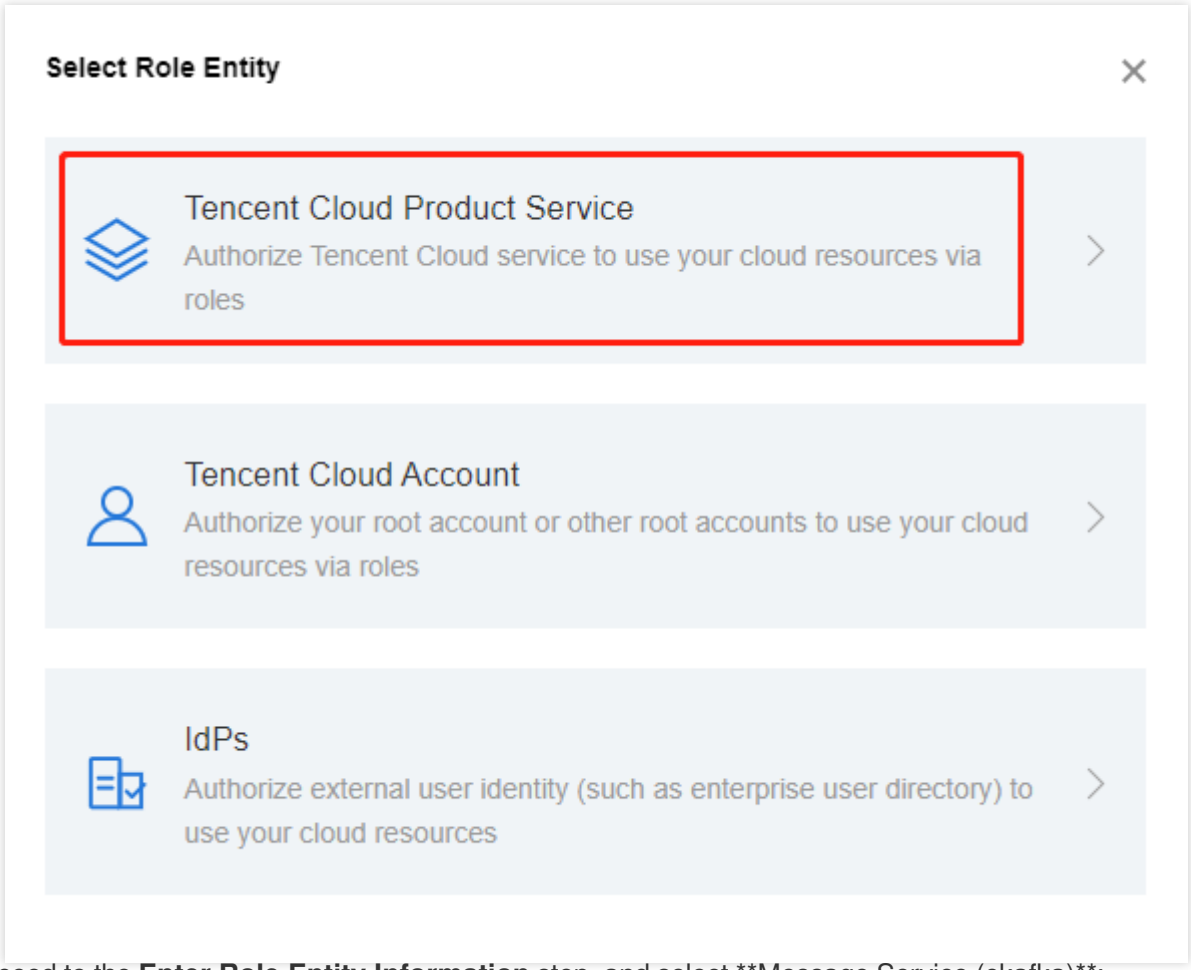
If the sub-account creating the connector task does not have the CAM policy permissions (QcloudCamRoleFullAccess), you may encounter prompts about missing CreateRole or AttachRolePolicy permissions. If your account does not yet have the Datahub_QcsRole role, see **Creating Role** for authorization instructions. If the account has the Datahub_QcsRole role, see **Authorizing Role** for authorization instructions.

Creating a Role

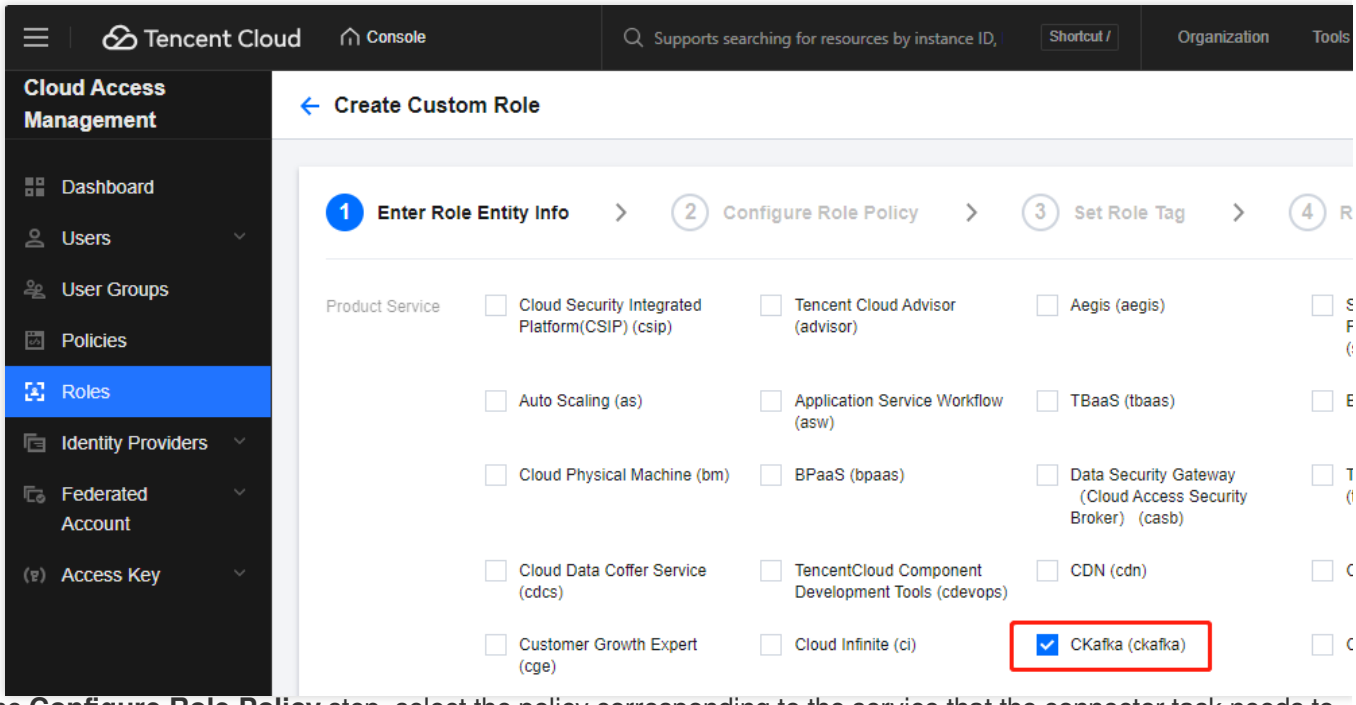
1. If you encounter a prompt about missing CreateRole policy permissions, a user with AdministratorAccess is required to go to the CAM console, enter the role page, and click **Create Role**.



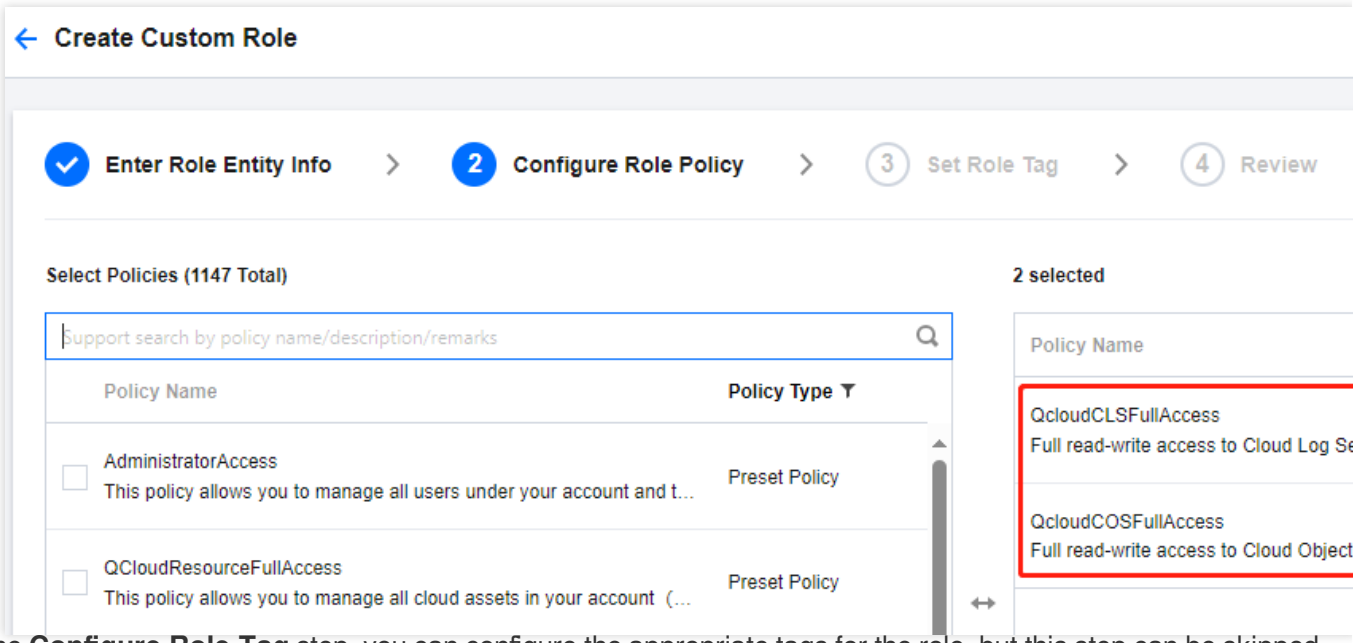
2. On the **Select Role Entity** page, select Tencent Cloud product service:



3. Proceed to the **Enter Role Entity Information** step, and select **Message Service (ckafka)**:

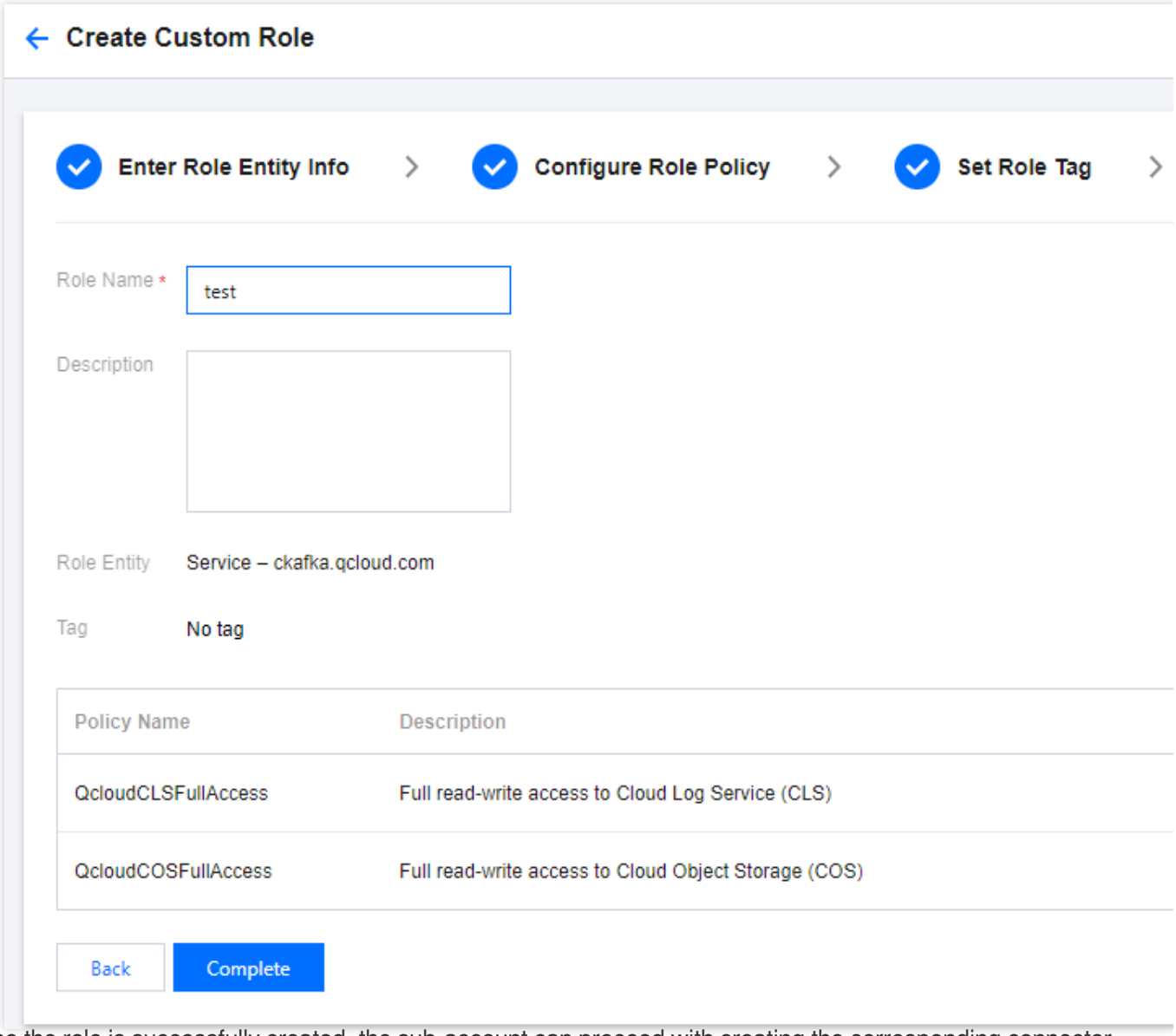


4. In the **Configure Role Policy** step, select the policy corresponding to the service that the connector task needs to access. Here, the policies for CLS and COS are selected:



5. In the **Configure Role Tag** step, you can configure the appropriate tags for the role, but this step can be skipped.

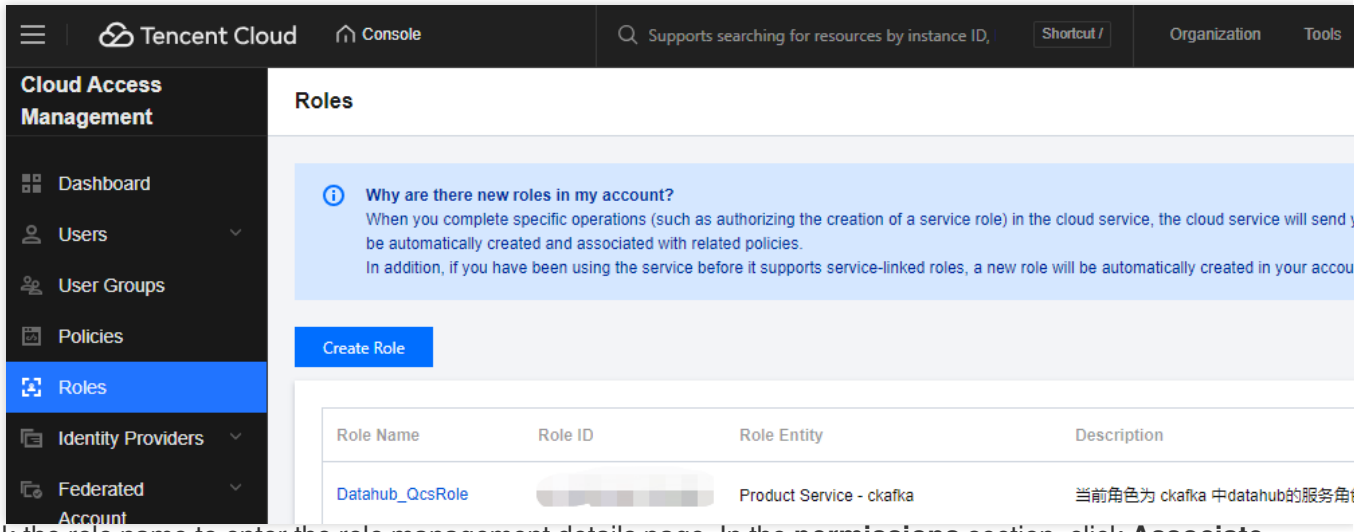
6. In the **Review** step, name the role as **Datahub_QcsRole**:



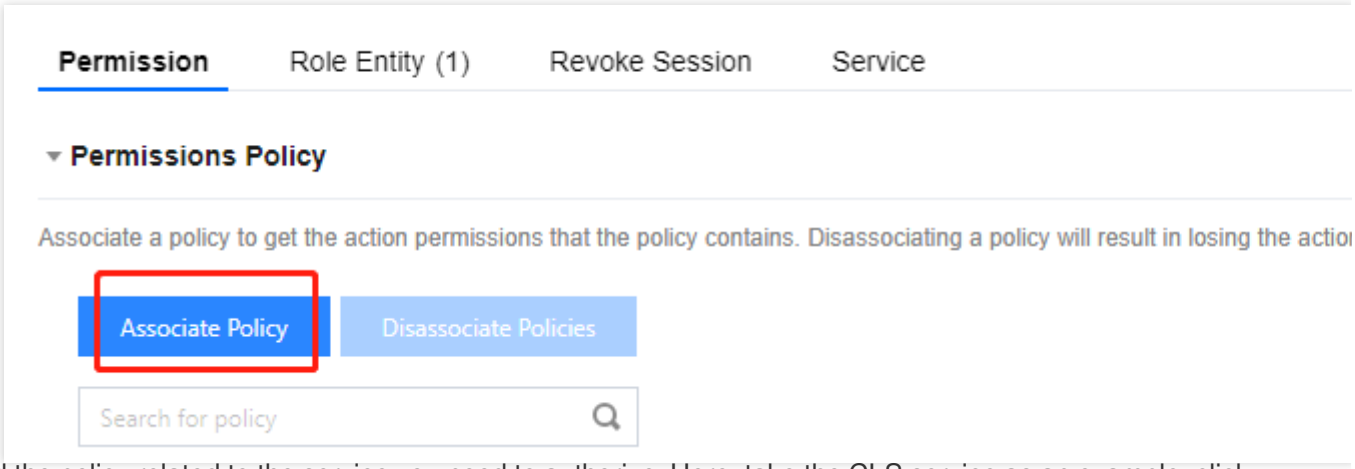
7. Once the role is successfully created, the sub-account can proceed with creating the corresponding connector tasks.

Authorize the Role

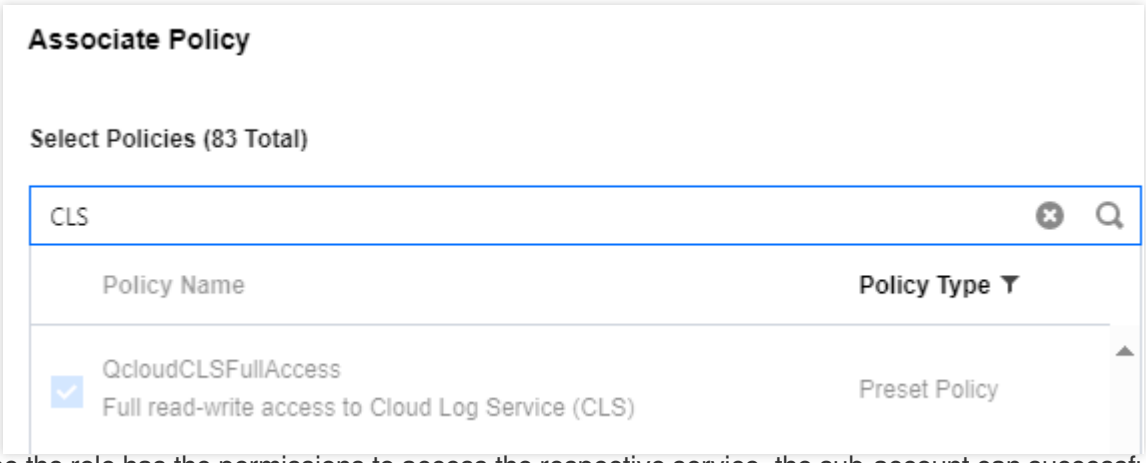
1. If you encounter a prompt about missing AttachRolePolicy policy permissions, a user with AdministratorAccess needs to go to the CAM console, enter the role page, and find the role corresponding to the service. Here, the Datahub_QcsRole role is taken as an example.



2. Click the role name to enter the role management details page. In the **permissions** section, click **Associate Policy**:



3. Find the policy related to the service you need to authorize. Here, take the CLS service as an example, click **Confirm** to complete the authorization:



4. Once the role has the permissions to access the respective service, the sub-account can successfully create the corresponding connector tasks.

What Is a Signaling Table

Last updated：2024-09-09 21:47:20

Background

The debezium connector initially synchronizes the existing data of the table only when a connection task is created. Subsequent newly added tables cannot trigger the synchronization of existing data. To support the synchronization of existing data for newly added tables, debezium uses a signal mode to notify the connector to trigger the synchronization of existing data for newly added tables.

Principles

You need to create a signaling table in the subscribed database. When you want to trigger the synchronization of existing data for a new table, insert the relevant information into the signaling table. Additionally, the connector needs to subscribe to this signaling table. When the connector receives a message from the signaling table, the synchronization of existing data for newly added tables will be triggered.

Notes

1. Since the subscription to the signaling table (dip_signal_taskId) messages has been added, the target Topic will include messages from the signaling table, which will require the business side to perform appropriate filtering.
2. Ensure that the user configured in Connection Management has the permissions to create, modify, and delete tables in the database. (This is only necessary for operations on the signaling table.)
3. If the signaling table is used to synchronize existing data for a newly added table, there may be instances of duplicate incremental data. To avoid duplicate data, you can pause the insert, update, and delete operations on the new table, edit the connector task's data source to add the table, and then resume the operations on the new table. Alternatively, you can perform idempotent processing downstream.
4. If you did not select the option to synchronize existing data when the task is created, and later select the option when you modified the data source to add a table with no change messages generated during this period, Debezium will synchronize the existing data of both the old and new tables (not related to the signaling table) by default. If there were incremental change messages in the old table before modifying the data source, then when you modify the data source to add a table and select the existing data synchronization, only the existing data of the new table will be synchronized (triggered by the signaling table).